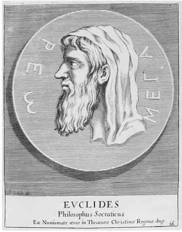
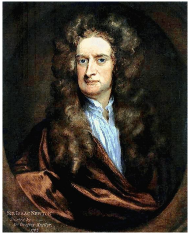
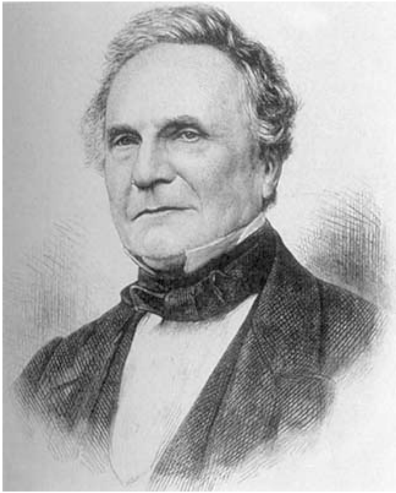
19: Computing History

Babylonian cuneiform circa 2000 B.C. (Photo by Matt Neale)

- Euclid’s algorithm circa 300 B.C.
- Abu Ja’far Muhammad ibn Musa Al-Khwarizmi’s books on algebra and arithmetic computation using Indo-Arabic numerals, circa 800 A.D.
- Isaac Newton (1643-1727) hired a “computer” to help with his work (e.g. a human being performing computations)
- Katherine Johnson (1918-2020) - "Human computer" who did trajectory analysis for America’s first human spaceflight (1961).







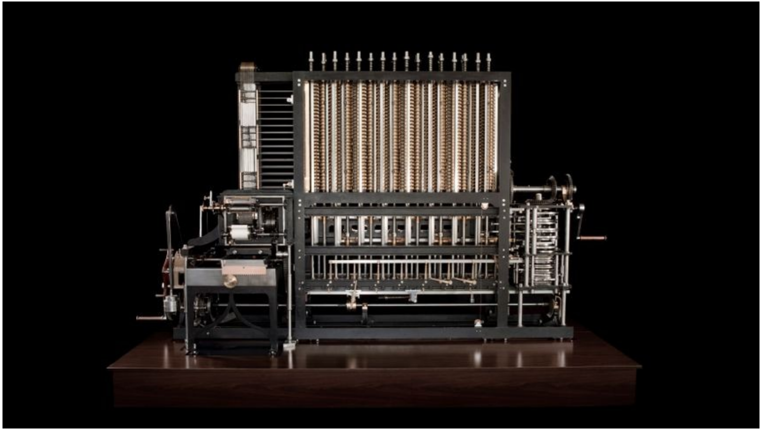
Developed mechanical computation for military applications:

- Difference Engine (1819)
- Analytical Engine (1834)

The specification of computational operations was separated from their execution

Babbage's designs were technically too ambitious

Video of a (modern) working model at `https://www.computerhistory.org/babbage/`.



https://www.computerhistory.org/babbage/

Assisted Babbage in explaining and promoting his ideas

Wrote articles describing the operation and use of the Analytical Engine

The first computer scientist?

Formalized the axiomatic treatment of Euclidean geometry

Hilbert's 23 problems (ICM, 1900)

Problem #2: Is mathematics consistent?

Axiom: A statement accepted without proof. For example, $\forall n : \quad n + 0 = n$.

Proposition: A statement we'd like to prove. For example, "The square of any even number is even."

Formula: A statement expressed with an accepted set of symbols and syntax. For example, $\forall n (\exists k : n = k + k \Rightarrow \exists m : m + m = n * n)$

Proof: A finite sequence of axioms (basic true statements) and accepted derivation rules (e.g. ϕ and $\phi \rightarrow \sigma$ yield σ).

Theorem: A mathematical statement ϕ together with a proof deriving ϕ.

> Hilbert's questions (1920's)

- Is mathematics complete? Meaning: for any formula ϕ, if ϕ is true, then ϕ is provable.
- Is mathematics consistent? Meaning: for any formula ϕ, there aren't proofs of both ϕ and $\neg\phi$.
- Is there a procedure to, given a formula ϕ, produce a proof of ϕ, or show there isn't one?

Hilbert believed the answers would be "yes".

Kurt Gödel (1906-78)

Gödel's answers to Hilbert (1929-30):

- Any axiom system powerful enough to describe arithmetic on integers is not complete.
- If it is consistent, its consistency cannot be proved within the system.

Define a mapping between logical formulas and numbers.

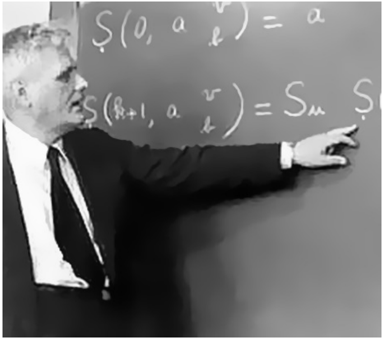
Use it to define mathematical statements saying "This number represents a valid formula", "This number represents a sequence of valid formulae", "This number represents a valid proof", "This number represents a provable formula".

Construct a formula ϕ represented by a number *n* that says "The formula represented by *n* is not provable". The formula ϕ cannot be false, so it must be true but not provable.

- Is there a procedure which, given a formula ϕ, either proves ϕ, shows it false, or correctly concludes ϕ is not provable?

The answer to this requires a precise definition of "a procedure", in other words, a formal model of computation.

Alonzo Church (1903-1995)



Set out to give a final “no” answer to this last question

With his student Kleene, created notation to describe functions on the natural numbers.

They wanted to modify Russell and Whitehead's notation for the class of all x satisfying a predicate f: $\hat{x} f(x)$.

But their notion was somewhat different, so they tried putting the caret before: $\hat{\ } x$.

Their typewriter could not type this, but had Greek letters.

Perhaps a capital lambda? Λx.

Too much like the symbol for logical AND: $\wedge$.

Perhaps a lower-case lambda? λx.

Example	Lambda calculus	Racket
The function that adds 2 to its argument:	$\lambda x.x+2$	`(lambda (x) (+ x 2))`
The function that subtracts its second argument from its first:	$\lambda x.\lambda y.x-y$	`(lambda (x)` `   (lambda (y) (- x y)))`
Function application:	fx	`(f x)`
Function application (left associativity):	fxy	`((f x) y)`

To prove something is impossible to express in some notation, the notation should be as simple as possible.

To make things even simpler, the lambda calculus did not permit naming of functions (only parameters), naming of constants like 2, or naming of functions like +.

It had three grammar rules and one reduction rule (function application).

How could it say anything at all?

$0 \equiv \emptyset$ or $\{\}$ (the empty set)

$1 \equiv \{\emptyset\}$

$2 \equiv \{\{\emptyset\}, \emptyset\}$

In general, n is represented by the set containing the sets representing $n-1, n-2, \ldots, 0$.

This is the way that arithmetic can be built up from the basic axioms of set theory.

$0 \equiv \lambda f.\lambda x.x$	the function which ignores its argument and returns the identity function	`(lambda (f)` `  (lambda (x) x))`
$1 \equiv \lambda f.\lambda x.fx$	the function which, when given as argument a function f, returns the same function	`(lambda (f)` `  (lambda (x) (f x)))`
$2 \equiv \lambda f.\lambda x.f(fx)$	the function which, when given as argument a function f, returns f composed with itself or $f \circ f$	`(lambda (f)` `  (lambda (x) (f (f x))))`

In general, n is the function which does n-fold composition.

With some care, one can write down short expressions for the addition and multiplication functions.

Similar ideas will create Boolean values, logical functions, and conditional expressions.

General recursion without naming is harder, but still possible.

The lambda calculus is a general model of computation.

Church proved that there was no computational procedure to tell if two lambda expressions were equivalent (represented the same function).

His proof mirrored Gödel’s, using a way of encoding lambda expressions using numbers, and provided a “no” answer to the idea of deciding provability of formulae.

This was published in 1936.

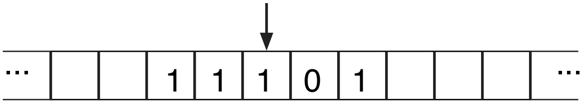
Independently, a few months later, a British mathematician came up with a simpler proof.

Turing defined a different model of computation, and chose a different problem to prove uncomputable.

This resulted in a simpler and more influential proof.

```
;; A Move is one of 'left, 'none, 'right.

;; (f state ch) produces a new state, a character to write on the tape at
;;      the current head position, and a head motion
;; f: State Char → State Char Move
```



Finite state control plus unbounded storage tape

Turing showed how to implement the controller (`f`) using characters. He gave several examples, including computing `01010101...` and `001011011101111011111011111110....`

Turing showed how to encode a function, `f`, so that it can be placed on the tape along with its data, x. He then showed how to write a different function, `u`, so that `(u f x)` $\equiv$ `(f x)` (for any `f`). He called `u` "the universal computing machine".

He then assumed that there was a machine that could process such a description and tell whether the coded machine would halt (terminate) or not on its input.

Using this machine, one can define a second machine that acts on this information.

The second machine uses the first machine to see if its input represents a coded machine which halts when fed its own description.

If so, the second machine runs forever; otherwise, it halts.

Feeding the description of the second machine to itself creates a contradiction: it halts iff it doesn't halt.

So the first machine cannot exist.

Turing's proof also demonstrates the undecidability of proving formulae.

Turing's ideas can be adapted to give a similar proof in the lambda calculus model.

Upon learning of Church's work, Turing quickly sketched the equivalence of the two models.

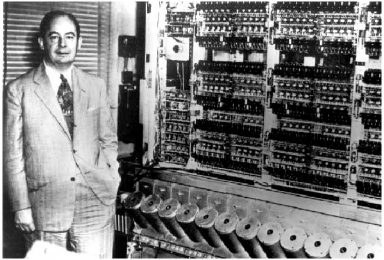
Turing's model bears a closer resemblance to an intuitive idea of real computation.

It would influence the future development of hardware and thus software, even though reasoning about programs is more difficult in it.

Turing went to America to study with Church at Princeton, earning his PhD in 1939.

During World War II, he was instrumental in an effort to break encrypted German radio traffic. Co-workers developed what we now know to be the world's first working electronic computer (Colossus).

Turing made further contributions to hardware and software design in the UK, to the field of artificial intelligence (the Turing test), and to pattern formation and mathematical biology before his untimely death in 1954.



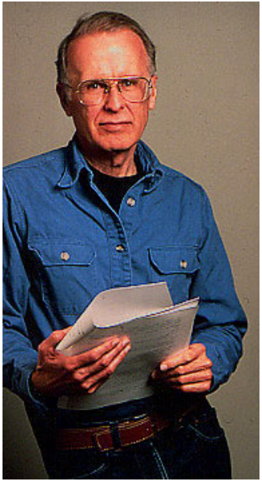
von Neumann was a founding member of the Institute for Advanced Study at Princeton.

In 1946 he visited the developers of ENIAC at the University of Pennsylvania, and wrote an influential “Report on the EDVAC” regarding its successor.

Features: random-access memory, CPU, fetch-execute loop, stored program.

Lacking: support for recursion (unlike Turing’s UK designs)

- Wrote the first compiler
- Defined first English-like data processing language, FLOW-MATIC, in the mid-1950's
- Many of her ideas were folded into COBOL (1959)



FORTRAN, designed by John Backus, was an early programming language influenced by architecture.

```
      INTEGER FN, FNM1, TEMP
      FN = 1
      FNM1 = 0
      DO 20 I = 1, 10, 1
      PRINT 10, I, FN
   10 FORMAT(I3, 1X, I3)
      TEMP = FN + FNM1
      FNM1 = FN
   20 FN = TEMP
```

John Backus (1924-2007) and FORTRAN (1957)

FORTRAN became the dominant language for numerical and scientific computation. Backus also invented a notation for language description that is popular in programming language design today.

Backus won the Turing Award in 1978, and used the associated lecture to criticize the continued dominance of von Neumann's architectural model and the programming languages inspired by it.

He proposed a functional programming language for parallel/distributed computation.

An alternative to imperative programming

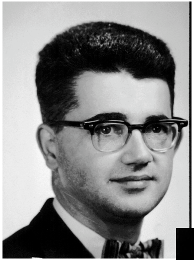
FORTRAN and COBOL, reflecting the Turing - von Neumann approach, dominated practical computing through most of the ’60’s and ’70’s.

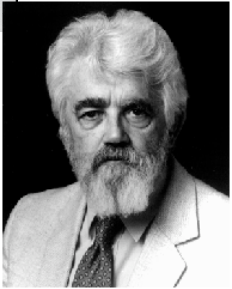
Many other computer languages were defined, enjoyed brief and modest success, and then were forgotten. The C programming language is an exception. It was introduced in 1972 and still enjoys widespread use. It is used in CS136.

Church’s work proved useful in the field of operational semantics, which sought to treat the meaning of programs mathematically.

It also was inspirational in the design of a still-popular high-level programming language called Lisp.

John McCarthy (1927-2011)





John McCarthy, an AI researcher at MIT, was frustrated by the inexpressiveness of machine languages and the primitive programming languages arising from them (no recursion, no conditional expressions).

In 1958, he designed and implemented Lisp (LISt Processor), taking ideas from the lambda calculus and the theory of recursive functions.

His 1960 paper on Lisp described the core of the language in terms that CS135 students would recognize.

McCarthy defined these primitive functions: `atom` (the negation of `cons?`), `eq`, `car` (`first`), `cdr` (`rest`), and `cons`.

He also defined the special forms `quote`, **`lambda`**, **`cond`**, and `label` (**`define`**).

Using these, he showed how to build many other useful functions.

The first implementation of Lisp, on the IBM 704, could fit two machine addresses (15 bits) into parts of one machine word (36 bits) called the address and decrement parts.

This led to the language terms `car` (`first` in Racket) and `cdr` (`rest` in Racket), which persist in Racket and Lisp to this day.

Lisp quickly evolved to include proper numbers, input/output, and a more comprehensive set of built-in functions.

Lisp became the dominant language for artificial intelligence implementations.

It encouraged redefinition and customization of the language environments, leading to a proliferation of implementations.

It also challenged memory capabilities of 1970's computers, and some special-purpose "Lisp machines" were built.

Modern hardware is up to the task, and the major Lisp groups met and agreed on the Common Lisp standard in the 1980's.

Starting about 1976, Carl Hewitt, Gerald Sussman, Guy Steele, and others created a series of research languages called Planner, Conniver, and Schemer (except that “Schemer” was too long for their computer’s filesystem, so it got shortened to “Scheme”).

Research groups at other universities began using Scheme to study programming languages.

Sussman, together with colleague Hal Abelson, started using Scheme in the undergraduate program at MIT. Their textbook, “Structure and Interpretation of Computer Programs” (SICP) is considered a classic.

The authors of the “How to Design Programs” textbook that formed the basis for CS135 developed an extension of Scheme (PLT Scheme) and its learning environment (DrScheme) to remedy the following perceived deficiencies of SICP:

- lack of programming methodology
- complex domain knowledge required
- steep, frustrating learning curve
- insufficient preparation for future courses

As PLT Scheme and the teaching languages diverged further from Sussman and Steele’s Scheme, they renamed their language Racket in 2010.

Imperative and functional convergence?

Languages are becoming more **multi-paradigm**, allowing programmers to think and code in the programming style that best suits their particular problem.

Some languages that started as primarily imperative or object-oriented are gaining functional aspects. These include C++, C#, Java, Go, and Python.

"Recently" defined languages are often multi-paradigm from the beginning. These include:

- Scala
- Kotlin
- Ruby
- JavaScript

Ex. 1

Using the years of the following computer history events as the keys, draw a Binary Search Tree that is the result of inserting the following in the order listed:

- Design of Babbage's difference engine
- Godel's incompleteness theorems
- Invention of Scheme
- Invention of COBOL
- Invention of Euclid's Algorithm
- Invention of FORTRAN
- Report on the EDVAC
- Design of Babbage's analytical engine
- Hilbert's 23 problems
- Invention of LISP
- Church's undecidability theorem

Goals of this module

- You should understand that important computing concepts pre-date electronic computers.
- You should understand, at a high level, the contributions of pioneers such as Babbage, Ada Augusta Byron, Hilbert, Church, Turing, Gödel, and others.
- You should understand the relationship between Church's work and functional programming as well as the relationship between Turing's work and imperative programming.

With only a few language constructs (**`define`**, **`cond`**, **`define-struct`**, `cons`, **`local`**, **`lambda`**) we have described and implemented ideas from introductory computer science.

We have done so without many of the features (static types, mutation, I/O) that courses using conventional languages have to introduce on the first day. The ideas we have covered carry over into languages in more widespread use.

We hope you have been convinced that a goal of computer science is to implement useful computation in a way that is correct and efficient as far as the machine is concerned, but that is understandable and extendable as far as humans are concerned.

These themes will continue in CS136 with additional themes and a new programming language using a different paradigm.

We have been fortunate to work with very small languages (the teaching languages) writing very small programs which operate on small amounts of data.

In CS136, we will broaden our scope, moving towards the messy but also rewarding realm of the “real world”.

The main theme of CS136 is scalability: what are the issues which arise when things get bigger, and how do we deal with them?

- How do we organize a program that's bigger than a few screenfuls?
- How do we share code between programs?
- How do we design programs to run efficiently?
- How can we leverage types to discover errors early?
- Are there better ways to handle errors?
- When is it appropriate to abstract away from implementation details for the sake of the big picture, and when must we focus on exactly what is happening at lower levels for the sake of efficiency?

These are issues which arise not just for computer scientists, but for anyone making use of computation in a working environment.

We can build on what we have learned this term to meet these challenges with confidence.