History

CS135 Lecture 22

Stories from our past

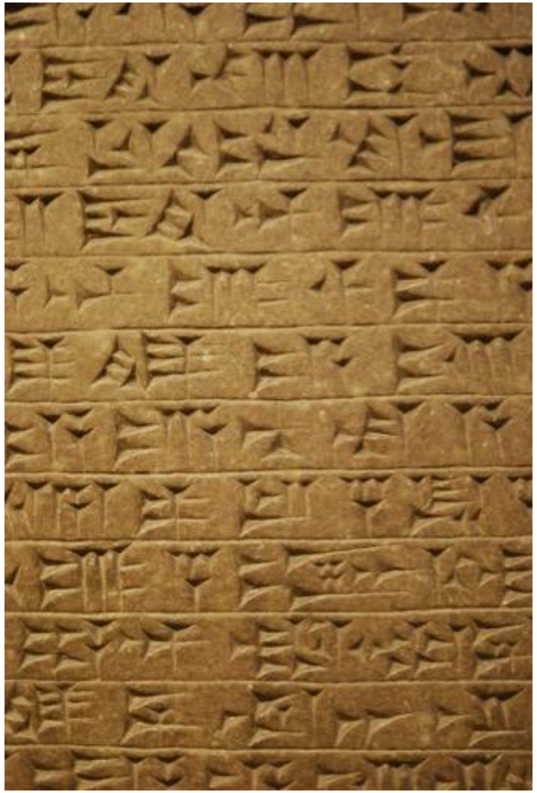
CS135 has traditionally included a lecture on the History of Computer Science near the end of term. However, this “history” is not a history in the historian’s sense of the word. Professional historians would be skeptical of the “great people and ideas” timeline in this lecture. They would want social, institutional, and cultural context, not just names and factoids.

This lecture is closer to a mythology of computer science: A collection of stories we tell ourselves about where we came from, with a special emphasis on the roots of the material covered in CS 135.

This lecture also is a product of our times, selecting and telling stories we now value from a Canadian cultural perspective in the early-to-mid 21st century.

L22.0 Calculation

The birth of mathematics



Computation, in the sense we now use the term, has its roots in the need for accurate calculation for practical purposes such as inventories, expenditures, and tax records.

Babylonian cuneiform circa 2000 BCE, showing mathematical notation in base 60. (Photo credit: Matt Neale)

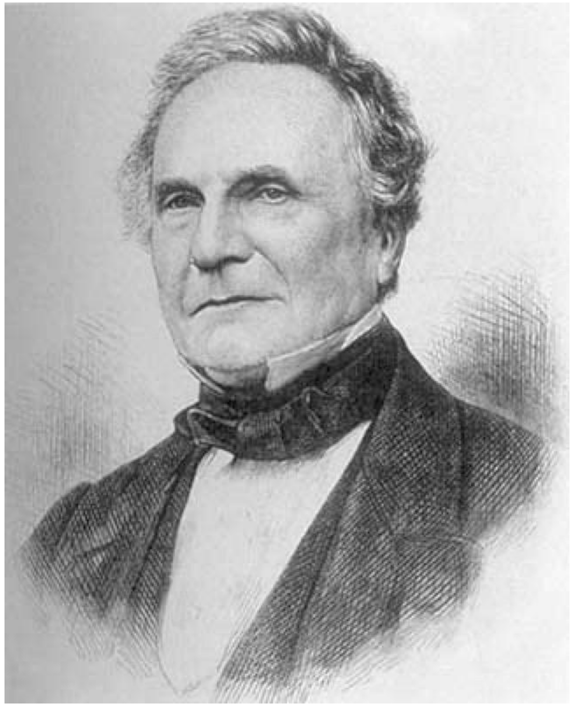
Early algorithms

You have already seen Euclid’s algorithm, the sieve of Eratosthenes, and other early algorithms.

Using an algorithm based on inscribing polygons in circles, the Chinese mathematician Liu Hui (劉徽) determined the value of π to six decimal places (3.14159) in 263 CE, which is sufficient accuracy for almost any practical purpose.

Abu Ja’far Muhammad ibn Musa Al-Khwarizmi (محمد بن موسى الخوارزمّي) wrote books on algebra and arithmetic computation using Indo-Arabic numerals, circa 800 A.D. The word “algorithm” comes from his name.

Charles Babbage (1791-1871)



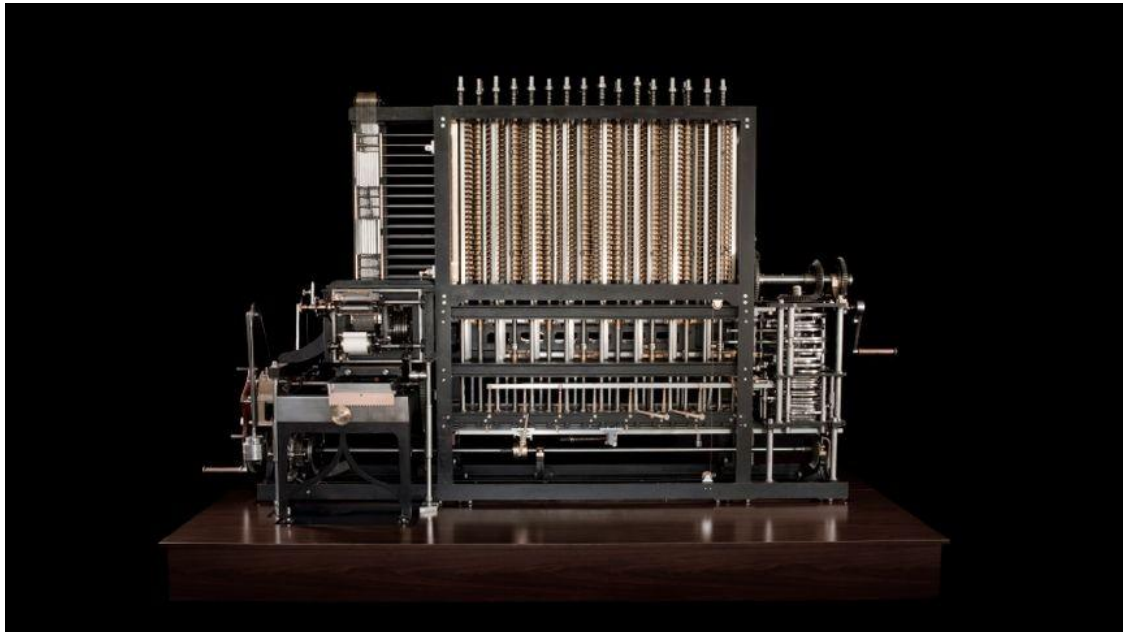
Babbage developed speculative mechanical computers for military applications:

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

The specification of their computational operations was separated from their execution. Unfortunately, Babbage's designs were technically too ambitious for their time.

There's a video of a modern working model at https://www.computerhistory.org/babbage/.

Babbage’s difference engine



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

Ada Augusta Byron (1815-1852)

Byron was the daughter of the poet Lord Byron and the wife of the Count of Lovelace, hence an aristocrat by both birth and marriage. Mathematically talented, she was tutored by Augustus de Morgan.

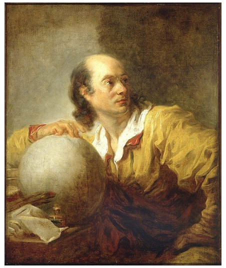
Byron assisted Babbage in explaining and promoting his ideas, and wrote articles describing the operation and use of the Analytical Engine

Often called the “first computer programmer”, she placed importance on communication and conceptual understanding.

Human computers

Until the emergence of purely electronic computers, a "computer" was a job. Computers were responsible for accurately executing complex calculations for scientific, military and other purposes.

One of the earliest examples is the computation of the orbit of Halley's comet by Alexis-Claude Clairaut (1713–1765), who recruited Jerome Lalande (1732–1807) and Nicole-Reine Lepaute (1723–1788) as human computers. Working together for five months during 1757 they correctly determined that the comet would reach closest approach to the sun on April 13, 1759.



Human computers

During and after World War II, human computers were often women holding degrees in mathematics, working with mechanical or early electronic calculators. Many of the computers who worked on the Manhattan Project, and later for NASA in the 1950s and 1960s, were women.

Katherine Johnson (1918–2020) started her career at NASA as a human computer, but quickly moved into the Flight Research Division to work as a research mathematician. She performed trajectory analysis for America’s first human spaceflight (1961), as portrayed in the 2016 film *Hidden Figures*.

L22.1 The death of mathematics?

David Hilbert (1862-1943)

One of the most influential mathematicians of all time, Hilbert famously collected 23 unsolved problems for a keynote address at the 1900 International Congress of Mathematicians. Several are still unsolved, of which the best known is the “Riemann hypothesis”.

One of these problems asked if mathematics is *consistent.* Attempts to solve this problem, together with Hilbert’s other work on logic, grew into a program to put all of mathematics on a formal basis, which may be summarized by three questions.

Hilbert’s questions

- *Is mathematics complete?* Completeness means that for any statement or proposition φ, if φ is true, then φ is provable.
- *Is mathematics consistent?* Consistency means that for any statement φ, there aren’t proofs of both φ and ¬φ (i.e., not φ) .
- *Is mathematics decidable?* Given a statement φ, is there a procedure to either produce a proof of φ or show there isn’t one?

By “statement” we mean a well-formed formula in some system of logic that has a definite truth value.

Hilbert believed the answer to all three questions is “yes”.

Kurt Gödel (1906-78)

In work completed by 1931, when Gödel was 25, he proved that:

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

Gödel's incompleteness theorems answered the first two of Hilbert's questions. The answer was "no".

Sketch of Gödel’s proof

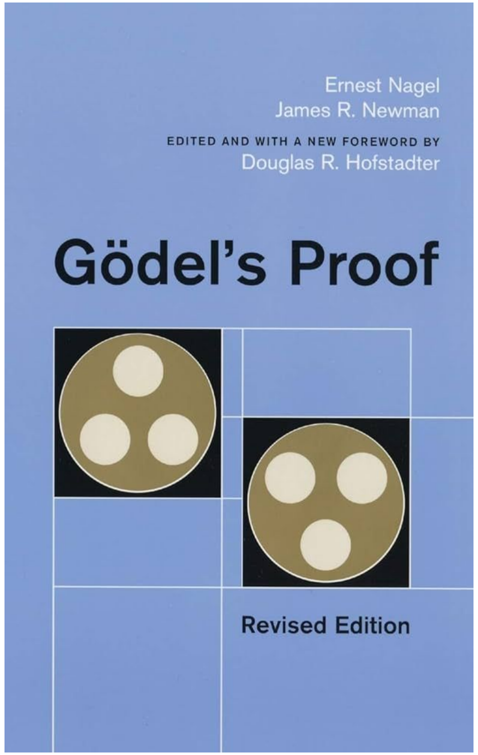
Define a mapping between logical formulas and numbers.

Use it to define mathematical statements such as:

- “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.

Gödel’s proof



If you’re interested in a more complete sketch, the book *Gödel’s proof* by Ernest Nagel and James R. Newman is an accessible treatment that’s understandable at the level of first-year CS/Math.

If you really want to understand Gödel’s proof you can take PMATH 432/632 (“First order logic and computability”) where several weeks are devoted to the proof.

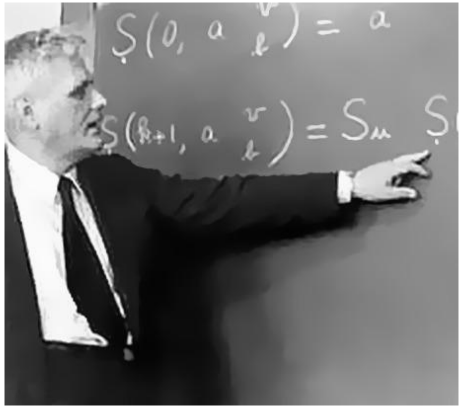
Hilbert's questions

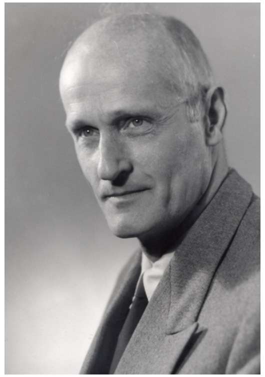
- ~~*Is mathematics complete?* Completeness means that for any statement or proposition φ, if φ is true, then φ is provable.~~
- ~~*Is mathematics consistent?* Consistency means that for any statement φ, there aren't proofs of both φ and ¬φ (i.e., not φ) .~~
- *Is mathematics decidable?* Given a statement φ, is there a procedure to either produce a proof of φ or show there isn't one?

Answering this last question requires a precise definition of "a procedure", in other words, a formal model of computation. That formal model is one that will seem familiar to you.

L22.2 Lambda Calculus

Alonzo Church (1903-1995)





Church set out to give a final “no” answer to the last of Hilbert’s questions. With his student Stephen Kleene (1909-1994), he created notation to describe functions on the natural numbers.

Kleene did further work in CS, including in the theory of regular expressions, which you will see in CS 241 and CS 360. The star operator for repeating something an arbitrary number of times is sometimes known as “Kleene star”.

The Lambda Calculus

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)`

Numbers from sets

One set-theoretic construction of the natural numbers uses the empty set to represent zero and defining the *successor* of a number as $n + 1 = n \cup \{n\}$.

$0 \equiv \varnothing$ (or $\{\}$)

$1 \equiv \{\varnothing\}$

$2 \equiv \{\varnothing, \{\varnothing\}\}$

$3 \equiv \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$

John von Neumann (1903-1957) showed that this pattern could represent the natural numbers, providing a set-theoretic construction that models the “Peano axioms”.

Numbers from functions

$0 \equiv \lambda f.\lambda x.x$	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.

Numbers from functions

```
(define zero (λ (f) (λ (x) x)))
(define successor (λ (n) (λ (f) (λ (x) (f ((n f) x))))))

(define one   (successor zero))
(define two   (successor one))
(define three (successor two))
(define four  (successor three))

(check-expect ((two add1) 0) 2)
(check-expect ((three add1) 0) 3)
(check-expect ((two add1) ((three add1) 0)) 5)
```

General model of computation

As we know, expressions in λ-calculus are transformed and simplified through α-conversion and β-reduction.

With some care, one can encode short expressions for the addition and multiplication functions. Similar ideas create Boolean values, logical functions, and conditional expressions.

General recursion without naming is harder, but still possible, using a function you may have heard of: the *Y combinator*.

While the details are outside the scope of CS 135, the lambda calculus provides a general model of computation.

Church’s proof

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, providing a “no” answer: There is no procedure that, given a statement φ, can produce a proof of φ or show there isn’t one.

Church’s proof was published in 1936, well before modern electronic computers existed.

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

L22.3 The birth of computer science

Alan Turing (1912-1954)

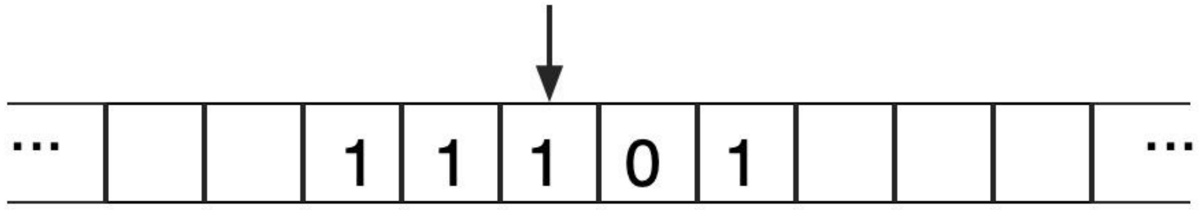
Turing defined a different model of computation and chose a different problem to prove uncomputable.

His model, later called a “Turing machine”, resulted in a simpler and more influential proof.

Turing’s model of computation

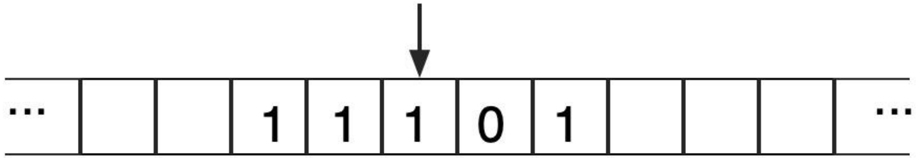
```
;; A Digit is (anyof 0 1)
;; A Direction is (anyof 'right 'left)
;; An Action is a (list Sym Digit Direction)

;; A Turing machine transition table (Machine) maps a
;; current state and the Digit under the head to an Action
;;
;; A Machine is a (listof (list (list Sym Digit) Action))
```



Example transition table

```
(define machine2
  (list
   (list (list 'A 0) (list 'B 1 'right))
   (list (list 'A 1) (list 'B 1 'left))
   (list (list 'B 0) (list 'A 1 'left))
   (list (list 'B 1) (list 'halt 1 'right))))
```



	A	B
0	1RB	1LA
1	1LB	1RH

Turing’s proof (1/2)

A Turing machine has finite state control plus unbounded storage tape. Turing gave several examples of these machines, including machines 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*) ≡ (*f x*) (for any *f*). He called *u* “the universal computing machine”, now called the “universal Turing 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.

Turing’s proof (2/2)

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 if and only if it doesn’t halt. So the first machine cannot exist.

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

Prof. Craig Kaplan provides a readable summary of Turing’s proof:
https://cs.uwaterloo.ca/~csk/halt/

Equivalence of Lambda Calculus and Turing Machines

Turing’s proof that the halting problem is undecidable can also be expressed in Lambda Calculus.

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, influencing the future development of hardware, and thus software.

The notion that any reasonable and general model of computation is equivalent (in a computability sense) to Turing and Church’s models is known as the “Church-Turing thesis”. A model of computation that is equivalent to a Turing machine is said to be “Turing complete”.

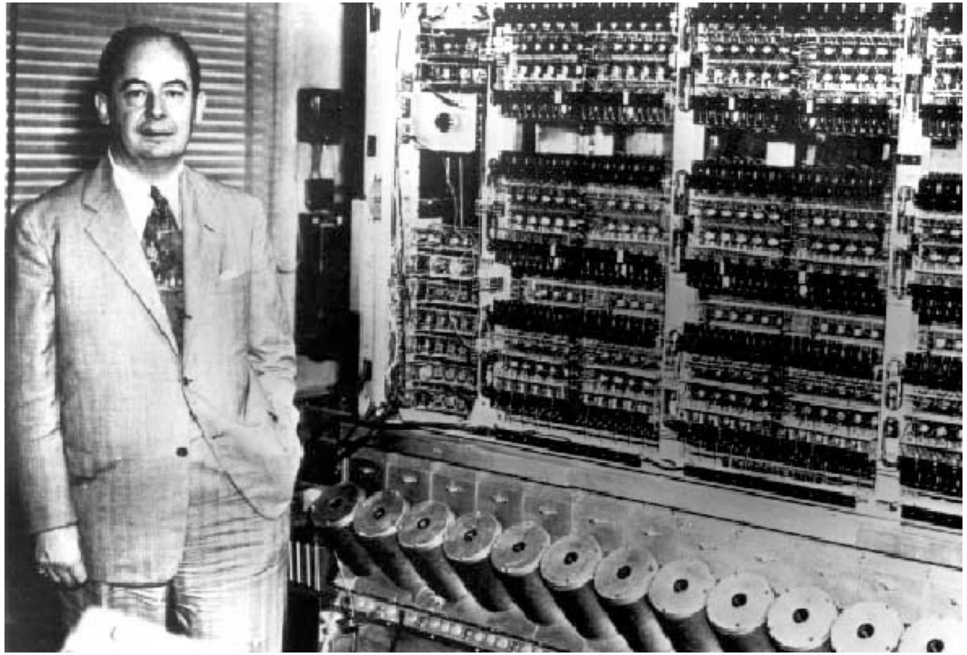
Turing’s legacy

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.

In 1950, Turing considered the question "Can machines think?", proposing a test of machine intelligence (now called “the Turing Test”) which asks an evaluator to engage in conversation with a person and a machine to determine which is which. The machine passes the test if the evaluator can’t tell.

Turing made further contributions, especially to hardware and software design in the UK, before his untimely death in 1954. The principal award for Computer Science research is named after him (“The Turing Award”).

John von Neumann (1903-1957)



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

In 1946 he visited the developers of ENIAC (pictured) at the University of Pennsylvania, and wrote an influential paper *First Draft of a Report on the EDVAC* describing its planned successor.

von Neumann’s legacy

von Neumann was among the greatest mathematician of the twentieth century, making fundamental contributions in a number of areas of mathematics and science (differential geometry, game theory, economics, quantum physics). He is sometimes credited with inventing mergesort.

The EDVAC report reads like a high-level description of the parts of a modern computer, featuring random-access memory, CPU, fetch-execute loop, and a stored program. To a large extent, the laptop in front of you is still based on a design called the “von Neumann architecture”.

The idea of a stored program computer (the program is data) is essential to modern computing, creating a desire to more effective programming methods…

L22.3 Programming

Programming languages

Initially computers were programmed primarily in “machine language”, low-level operations that directly referenced the physical hardware of a machine.

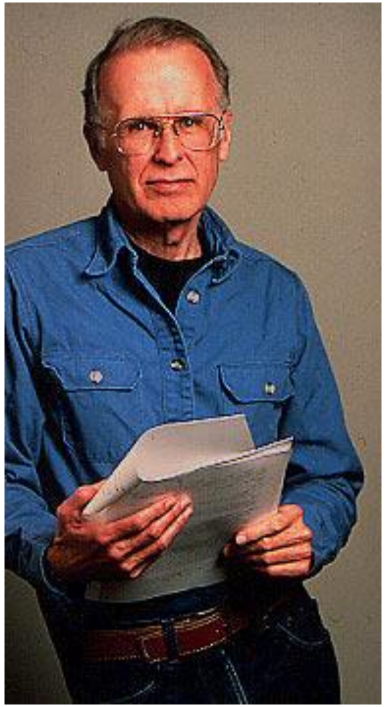
To make it faster to write computer code and to allow that code to easily be moved from machine to machine, higher-level programming languages began to be developed, which abstracted away many of the low-level hardware details. These programming languages were implemented by programs called “compilers”, which translated the high-level language into low-level operations. Nonetheless, these languages were often influenced by properties and limitations of hardware.

While many early programming languages have largely been forgotten, three of the earliest languages, FORTRAN, COBOL and Lisp, have had lasting legacies…

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

FORTRAN, designed by John Backus, quickly became the dominant language for numerical and scientific computation.

```
   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)

Backus also invented a notation for language description that had broad influence on programming language design (Backus–Naur form).

Backus received 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.

Nonetheless, FORTRAN continued to dominate scientific programming well into the 1990’s and is still used today. At their lowest-level, many Python numerical and statistical packages are written in FORTRAN.

WATFOR (1967)

Starting in the mid-1960s, the University of Waterloo became famous for its WATFOR and WATFIV compilers for the FORTRAN IV language.

Because they fit entirely in the computer’s main “core” memory, WATFOR and WATFIV were fast, and were also notable for the accuracy and completeness of their error diagnostics. They were ideal for teaching because many student programs could be run quickly, with understandable output.

WATFIV was widely used for teaching until at least the mid-1980s.

Through the 1970’s, the two CS courses required of all UW Math students taught them programming in FORTRAN and COBOL. Because of its focus on business computing, COBOL was recommended as a first course for those in co-op.

Grace Murray Hopper (1906-1992) and COBOL (1959)

Photo # NH 96919-KN Commodore Grace M. Hopper, 1984

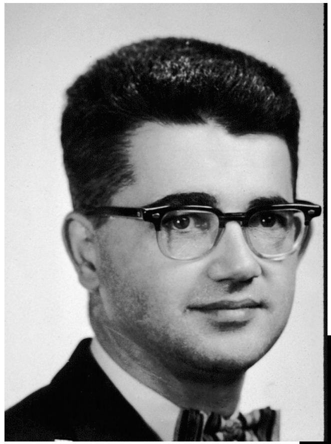
A professor at a New England college when World War II broke out, Hopper volunteered for the U.S. Navy and never left it, rising to the rank of rear admiral. Her main contribution was in making programs readable and writable by people without intimate knowledge of a particular machine architecture. The major conference on women in CS is named after her, as is an ACM award to young researchers.

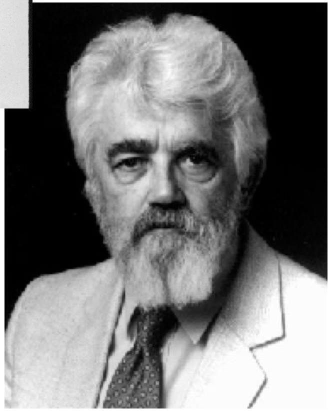
She wrote one of the first compilers. In the 1950s, she defined an English-like data processing language, FLOW-MATIC. Many of her ideas were incorporated into COBOL, which became one of the the most widely used programming language for decades.

John McCarthy (1927-2011) and Lisp (1958)

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

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





McCarthy’s Lisp

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.

From Lisp to Scheme to Racket

Lisp became the dominant language for artificial intelligence implementations. It encouraged redefinition and customization of the language environments, leading to a proliferation of implementations.

Starting about 1976, Carl Hewitt, Gerald Sussman, Guy Steele, and others created a series of research languages called Planner, Conniver, and Schemer (which was shortened to “Scheme”).

Sussman, together with colleague Hal Abelson, started using Scheme in the undergraduate program at MIT. Over the years, Scheme evolved as a teaching language, giving birth to Racket and book on which this version of CS135 was originally based: *How to Design Programs* (https://htdp.org/).

Imperative vs. Functional Programming

Historically, people divided programming languages into “imperative” languages (e.g., FORTRAN and COBOL) and “functional” languages (e.g., Lisp).

Imperative languages focus on the storage and manipulation of data, mirroring the operation of computer hardware. Functional languages focus on computing new values through the application of functions, which are themselves treated as first class values.

The “intermediate student” level of Racket is essentially a purely functional language, while the C language used in CS136 is essentially a purely imperative programming language. Most modern languages (Python, Scala, C++, Rust, etc.) combine elements of both imperative and functional programming styles.

The death of computer science?

In 2018 Yoshua Bengio, Geoffrey Hinton, and Yann LeCun, received the Turing Award for “conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing”.

Their work ultimately led to the creation of generative AI systems (e.g., ChatGPT) that can can not only pass the Turing Test, they can also effortlessly generate code, including the solutions to CS135 homework.

AI assistance has already fundamentally changed how software is created, with programmers increasingly employing AI to develop and test code.

Ultimately, AI assistance may allow us to create more complex and dynamic software systems, beyond anything we could realistically imagine a few years ago.

L22 Summary

L22: You should know

In the final exam we often ask a multiple choice question, or other simple question, related to this lecture. Generally these are questions about the “great people and ideas” we have talked about. You should recognize the names and have a sense of what they did, but you will not need to remember spellings of names, specific dates, or sketches of proofs.

If you are studying for the exam, read this lecture over, but don’t spend too much time on it.

Further reading

This lecture has focused on the “great people and ideas” behind the concepts you have learned in CS135.

If you would like a different version of this history, written by a professional historian, we suggest *The History of Computing: A Very Short Introduction* by Doron Swade, Oxford University Press, 2022.