Turing’s proof (1936): asking does it halt?

A “Turing machine” is equivalent to a Racket function, i.e. it consumes input and produces a result.

Turing assumed you could implement a machine (a function), (halt? m1 i) that consumed (1) a machine m1 and (2) an input i, it would determine if the m1 would halt when it consumed i.

He could then use halt? to implement a second function that would tell if a machine would halt or not when fed its own description as an input, i.e. (define (halt-on-myself? m2) (halt? m2 m2))
Using `halts-on-myself?`, one can define a machine (i.e. function) that acts on this information.

```
(define (m3 m2)
  (cond
    [(halts-on-myself? m2) (forever 1)]
    [else true]))
```

`m3` uses `halts-on-myself?` to see if `m2` represents a machine which halts when fed its own description.

If so, `m3` runs forever; otherwise, it halts.

When happens with `(m3 m3)`, i.e. `m3` consuming a copy of itself. It creates a contradiction: `m3` halts iff `m3` does not halt.
So the first machine \texttt{halts?} cannot exist.

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

\textbf{Key Point:} You do not have to understand this proof in this course.
Advantages of Turing’s ideas

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: other contributions

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

During the war, he was instrumental in an effort to break encrypted German radio traffic, resulting in the development of 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, and to the field of artificial intelligence, before his untimely death in 1954.
John von Neumann (1903-1957)
John von Neumann and EDVAC

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.

Key Point: The EDVAC contain many features of current computers: random-access memory, CPU, fetch-execute loop, stored program control.

Lacking: support for recursion (unlike Turing’s UK designs)
Grace Murray Hopper (1906-1992)

*Key Contributions:* Wrote the first *compiler*, defined the first *English-like data processing language* (early 1950’s) whose ideas later folded into COBOL (1959).
John Backus and FORTRAN (1957)

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 and FORTRAN

Key Contribution: In 1957 John Backus designed FORTRAN which 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 criticized 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.
FORTRAN, COBOL and Lisp

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.

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)
McCarthy and Functional Programming

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 CS 135 students would recognize.
McCarthy’s Lisp

Lisp was a forerunner to Racket. It introduced many of the ideas you see in Racket today.

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

He also defined the special forms \texttt{quote}, \texttt{lambda}, \texttt{cond}, and \texttt{label} (\texttt{define}).

Using these, he showed how to build many other useful functions.
The evolution of Lisp

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. Machine instructions facilitated such manipulation.

This led to the language terms car (i.e. first) and cdr (i.e. rest) 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.
Use of Lisp

**Key Point:** 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.
Computing Power in the 1960

- In 1961 (after two short term leases) U Waterloo obtained an IBM 1620.
  - It had 20 KB of memory.
  - It had 2 MB hard drive.
  - It could add or subtract two 5 digit numbers as a rate of 1,780/second.
  - It could multiply two 5 digit numbers a rate of 200/second.
  - This would average out to about 360 operates per second if the mix was 50% multiplication.
Scheme: a descendant of Lisp

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”).

Because of its simplicity, 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.
Scheme’s descendant: Racket

The authors of the HtDP textbook developed an extension of Scheme and its learning environment 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 it diverged further from Sussman and Steele’s Scheme, they renamed their language *Racket* in 2010.
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 origins of functional programming in Church’s work and the origins of imperative programming in Turing’s work.
Summing up CS 135

*Key Point:* 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 other humans are concerned.

These themes will continue in CS 136 with additional themes and a new programming language using a different paradigm.
Looking ahead to CS 136

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 CS 136, we will broaden our scope, moving towards the messy but also rewarding realm of the “real world”.

The main theme of CS 136 is *scalability*: what are the issues which arise when things get bigger, and how do we deal with them?
Looking ahead to CS 136

How do we *organize a program* that is bigger than a few screenfuls?

How do we *reuse and share code*, apart from cutting-and-pasting it into a new program file?

How do we design programs so that they *run efficiently*?

What changes might be necessary to our notion of types and to the way we *handle errors* when there is a much greater distance in time and space between when the program is written and when it is run?
Looking ahead to CS 136

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 in order to meet these challenges with confidence.
Module 16 Summary

Dawn of Computing

1. Computation goes back to some of the earliest writing. The earliest computers were people and the earliest algorithms were descriptions of how they could arrive at the result. [2-3]

2. Charles Babbage described the first device that was designed for general computation where the “program” was specified using punched cards. [4]

3. Ada Augusta Byron is considered by many to be the first computer scientist. She wrote articles describing algorithms for—and explaining how to use—Babbage’s Analytic Engine. [5]
Module 16 Summary

Limits of Mathematics

4. 1900: David Hilbert asked about the completeness and consistency of mathematics and if a procedure for producing proofs is possible. [6-8]

5. 1929-30: Kurt Gödel answered the first two questions: powerful systems are not complete and consistency cannot be proved within the system. [9-10]

6. 1936: Alonzo Church introduced $\lambda$-calculus as a formal model of computation and showed that there was no computational method to tell if two $\lambda$ expressions were equivalent. [12-19]
Module 16 Summary
Computers and Programming Languages

7. 1936: Alan Turing came up with a simpler model of computation (the Turing machine) and showed there is no program that can always determine if another program will halt or not. [20-25]

8. 1946: John von Neumann wrote an influential article about the EDVAC which has many features of a modern computer, called the von Neumann architecture. [26-27]

9. Grace Murray Hopper wrote the first compiler and an English-like programming language which became the basis of COBOL (used in commercial and business applications). [28]
Module 16 Summary
Programming Languages

10. 1957: John Backus created FORTRAN the dominant language for numeric and scientific computation. [29-30]

11. The ideas of Turing, von Neumann, Hopper, and Backus greatly influenced the design of computers. [31]

12. 1958: John McCarthy created Lisp (LISt Processor) taking ideas from λ-calculus. It became the dominant language in Artificial Intelligence. [32-36]
Module 16 Summary

Programming Languages and Programming Issues

13. Adaptations of Lisp to make it easier to learn in include Scheme (by Gary Sussman and Guy Steele in the 1970s) and then Racket (by Felleisen, Findler, Flatt, Krishnamurthi in the 1990s). [37-38]

14. Looking ahead to CS136 (and beyond) we will consider such issues as scalability (larger data sets), organizing large programs, efficiency and dealing with errors. [42-44]
Preparing for the final

• I will make a complete copy of all my slides (in one big file) available on the course website in the next few days.

• We have a Final Exam [official] post in Piazza giving some suggestions on how to study, listing which slides have been skipped, extra office hours before the final, etc.

• We are having help sessions before the final (two in classrooms and live one on the web).

• We will be monitoring and answering questions in Piazza.