There are many other models of computation, each of which describes a class of integer functions where a computation is performed on an integer input to produce an integer answer.

Kleene, Post and Turing were all students of Alonzo Church at Princeton University in the mid-1930's.

They explored many other formulations for modeling computation: recursive functions, rule-based systems for string rewriting, and the lambda calculus.

They were all particularly intrigued with proving the existence of problems unsolvable by realizable machines.

Which, of course, meant characterizing the problems that could be solved by realizable machines.

It turned out that each model was capable of computing *exactly* the same set of integer functions!

This was proved by coming up with constructions that translated the steps in a computation between the various models.

It was possible to show that if a computation could be described by one model, an equivalent description exists in the other model.

This lead to a notion of computability that was independent of the computation scheme chosen.

This notion is formalized by Church's Thesis, which says that every discrete function computable by any realizable machine is computable by some Turing Machine.

So if we say the function f(x) is computable, that's equivalent to saying that there's a TM that given x as an input on its tape will write f(x) as an output on the tape and halt.

As yet there's no proof of Church's Thesis, but it's universally accepted that it's true.

In general "computable" is taken to mean "computable by some TM".

If you're curious about the existence of uncomputable functions, please see the optional video at the end of this lecture.

Okay, we've decided that Turing Machines can model any realizable computation.

In other words for every computation we want to perform, there's a (different) Turing Machine that will do the job.

But how does this help us design a general-purpose computer?

Or are there some computations that will require a special-purpose machine no matter what?

What we'd like to find is a universal function U: it would take two arguments, k and j, and then compute the result of running T_k on input j.

Is U computable, i.e., is there a universal Turing Machine T_U?

If so, then instead of many ad-hoc TMs, we could just use T_U to compute the results for any computable function.

.249 Surprise!

U is computable and T_U exists.

If fact there are infinitely many universal TMs, some quite simple.

The smallest known universal TM has 4 states and uses 6 tape symbols.

A universal machine is capable of performing any computation that can be performed by any TM!

What's going on here?

k encodes a "program" - a description of some arbitrary TM that performs a particular computation.

j encodes the input data on which to perform that computation.

T_U "interprets" the program, emulating the steps T_k will take to process the input and write out the answer.

The notion of interpreting a coded representation of a computation is a key idea and forms the basis for our stored program computer.

The Universal Turing Machine is the paradigm for modern general-purpose computers.

Given an ISA we want to know if it's equivalent to a universal Turing Machine.

If so, it can emulate every other TM and hence compute any computable function.

How do we show our computer is Turing Universal?

Simply demonstrate that it can emulate some known Universal Turing Machine.

The finite memory on actual computers will mean we can only emulate UTM operations on inputs up to a certain size but within this limitation we can show our computer can perform any computation that fits into memory.

As it turns out this is not a high bar: so long as the ISA has conditional branches and some simple arithmetic, it will be Turing Universal.

This notion of encoding a program in a way that allows it to be data to some other program is a key idea in computer science.

We often translate a program Px written to run on some abstract high-level machine (eg, a program in C or Java) into, say, an assembly language program Py that can be interpreted by our CPU.

This translation is called compilation.

Much of software engineering is based on the idea of taking a program and using it as as component in some larger program.

Given a strategy for compiling programs, that opens the door to designing new programming languages that let us express our desired computation using data structures and operations particularly suited to the task at hand.

So what have learned from the mathematicians' work on models of computation?

Well, it's nice to know that the computing engine we're planning to build will be able to perform any computation that can be performed on any realizable machine.

And the development of the universal Turing Machine model paved the way for modern stored-program computers.

The bottom line: we're good to go with the Beta ISA!