MA/CSSE 474
Theory of Computation

Delayed due dates for HWs 12-14.
See updated schedule page.
No class meeting on Thursday.

Universal Turing Machine
Church-Turing Thesis

Recap: The Universal Turing Machine

Problem: All our machines so far are hardwired.

Question: Can we build a programmable TM that accepts as input:

M's description *input string*

executes the program, and outputs:

the *output string* that M would produce when started with the same input string

The Universal Turing Machine

To define the Universal Turing Machine *U* we need to:

1. Define an encoding operation for TMs.
2. Describe the operation of *U* given input <*M*, *w*>, the encoding of:
   - a TM *M*, and
   - an input string *w*.

Recap: An Encoding Example

Consider $M = (\{s, q, h\}, \{a, b, c\}, \{❑, a, b, c\}, \delta, s, \{h\})$:

state	symbol	δ
s	❑	(*q*,❑, →)
s	a	(*s*,b,→)
s	b	(*q*,a, ←)
s	c	(*q*,b, ←)
q	❑	(*s*,a, →)
q	a	(*q*,b,→)
q	b	(*q*,b, ←)
q	c	(*h*,a, ←)

state/symbol	representation
s	q00
q	q01
h	h10
❑	a00
a	a01
b	a10
c	a11

Do you have any questions about how this works?

<*M*> = (q00,a00,q01,a00,→), (q00,a01,q00,a10,→),
(q00,a10,q01,a01, ←), (q00,a11,q01,a10,←),
(q01,a00,q00,a01,→), (q01,a01,q01,a10,→),
(q01,a10,q01,a11,←), (q01,a11,h10,a01,←)

Enumerating Turing Machines

Theorem: There exists an infinite lexicographic enumeration of:

(a) All syntactically valid TMs.

(b) All syntactically valid TMs with specific input alphabet Σ.

(c) All syntactically valid TMs with specific input alphabet Σ and specific tape alphabet Γ.

Proof on next slide

Enumerating Turing Machines

Proof: Fix Σ = {`(`, `)`, `a`, `q`, `y`, `n`, `0`, `1`, comma, $\rightarrow$, $\leftarrow$}, ordered as listed. Then:

1. Lexicographically enumerate the strings in Σ^*.
2. As each string *s* is generated, check to see whether it is a syntactically valid Turing machine description. If it is, output it.

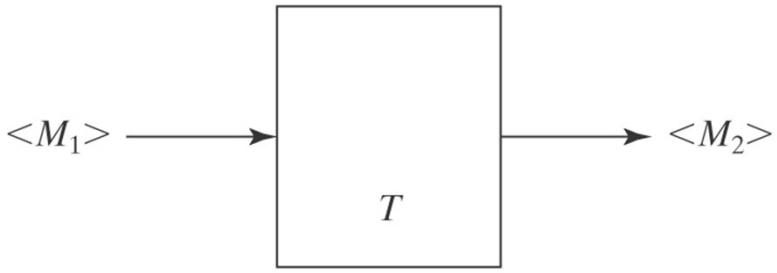
To restrict the enumeration to symbols in sets Σ and Γ, check, in step 2, that only alphabets of the appropriate sizes are allowed.

We can now talk about the i^{th} Turing machine.

Another Win of Encoding

One big win of defining a way to encode any Turing machine *M*:

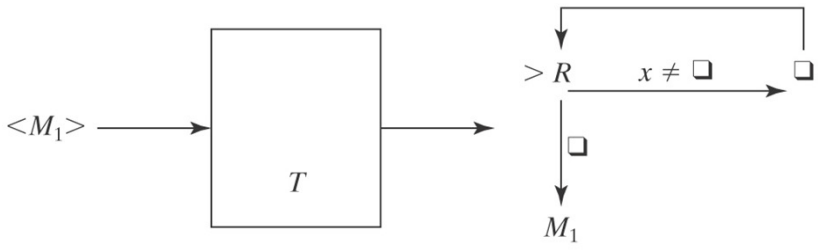
- We can talk about operations on programs (TMs).



Example of a Transforming TM *T*:

Input: (encoding of) a TM M_1 that reads its input tape and performs some operation *P* on it.

Output: (encoding of) a TM M_2 that performs *P* on an empty input tape.



Encoding Multiple Inputs

Let:

$<x_1, x_2, \ldots x_n>$

mean a single string that encodes the sequence of individual values:

$x_1, x_2, \ldots x_n$.

The Specification of the Universal TM

On input $<M, w>$, U must:

- Halt iff M halts on w.
- If M is a deciding or semideciding machine, then:
  - If M accepts, accept.
  - If M rejects, reject.
- If M computes a function, then $U(<M, w>)$ must equal $M(w)$.

How *U* Works

U will use 3 tapes:

- Tape 1: *M*'s tape.
- Tape 2: <*M*>, the "program" that *U* is running.
- Tape 3: *M*'s state.

The Universal TM

❑	<*M*----------			----------*M*,	*w*----------		----------*w*>	❑	❑
	1	0	0	0	0	0	0		
	❑	❑	❑	❑	❑	❑	❑		
	1	0	0	0	0	0	0		
	❑	❑	❑	❑	❑	❑	❑		
	1	0	0	0	0	0	0		

Initialization of *U*:

1. Copy <*M*> onto tape 2.
2. Look at <*M*>, figure out what *i* is, and write the encoding of state *s* on tape 3.

After initialization:

❑	❑	❑	❑	❑	<*w*----------		----------*w*>	❑	❑
	0	0	0	0	1	0	0		
	<*M*----------			----------*M*>	❑	❑	❑		
	1	0	0	0	0	0	0		
	q	0	0	0	❑	❑	❑		
	1	❑	❑	❑	❑	❑	❑		

The Operation of *U*

	❑	❑	❑	❑	<w---------w>				
	0	0	0	0	1	0	0		
❑	<M---------M>				❑	❑	❑	❑	❑
	1	0	0	0	0	0	0		
	q	0	0	0	❑	❑	❑		
	1	❑	❑	❑	❑	❑	❑		

Simulate the steps of *M* :

1. Until *M* would halt do:
    - 1.1 Scan tape 2 for a quintuple that matches the current state, input pair.
    - 1.2 Perform the associated action, by changing tapes 1 and 3. If necessary, extend the tape.
    - 1.3 If no matching quintuple found, halt. Else loop.
2. Report the same result *M* would report.

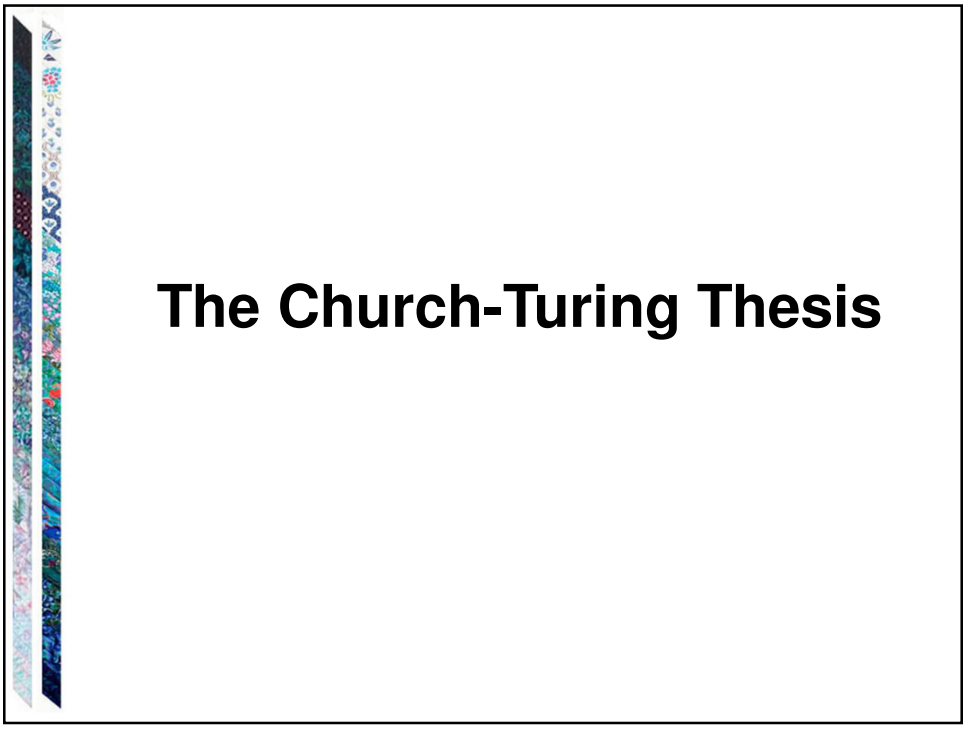
How long does *U* take?

If A Universal Machine is Such a Good Idea …

Could we define a Universal Finite State Machine?

Such a FSM would accept the language:

$L = \{<F, w> : F \text{ is a FSM, and } w \in L(F) \}$



Are We Done?

FSM $\Rightarrow$ PDA $\Rightarrow$ Turing machine

Is this the end of the line?

There are still problems we cannot solve with a TM:

- There is a countably infinite number of Turing machines since we can lexicographically enumerate all the strings that correspond to syntactically legal Turing machines.
- There is an uncountably infinite number of languages over any nonempty alphabet.
- So there are more languages than there are Turing machines.

What Can Algorithms Do?

1. Can we come up with a system of axioms that makes all true statements be theorems (I.e. provable from the axioms)?
   The set of axioms can be infinite, but it must be decidable

2. Can we always decide whether, given a set of axioms, a statement is a theorem or not?

In the early 20th century, it was widely believed that the answer to both questions was "yes."

Gödel's Incompleteness Theorem

Kurt Gödel showed, in the proof of his Incompleteness Theorem [Gödel 1931], that the answer to question 1 is no. In particular, he showed that there exists no decidable axiomatization of Peano arithmetic that is both consistent and complete.

Complete: All true statements in the language of the theory are theorems

The Entscheidungsproblem

From Wikipedia: The Entscheidungsproblem ("decision problem", David Hilbert 1928) asks for an algorithm that will take as input a description of a formal language and a mathematical statement in the language, and produce as output either "True" or "False" according to whether the statement is true or false. The algorithm need not justify its answer, nor provide a proof, so long as it is always correct.

Three equivalent formulations:

1. Does there exist an algorithm to decide, given an arbitrary sentence *w* in first order logic, whether *w* is valid?
2. Given a set of axioms *A* and a sentence *w*, does there exist an algorithm to decide whether *w* is entailed by *A*?
3. Given a set of axioms, *A*, and a sentence, *w*, does there exist an algorithm to decide whether *w* can be proved from *A*?

The Entscheidungsproblem

To answer the question, in any of these forms, requires formalizing the definition of an algorithm:

- Turing: Turing machines.
- Church: lambda calculus.

Turing proved that Turing machines and the lambda calculus are equivalent.

Church's Thesis (Church-Turing Thesis)

All formalisms powerful enough to describe everything we think of as a computational algorithm are equivalent.

This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

The Church-Turing Thesis

Examples of equivalent formalisms:

- Modern computers (with unbounded memory)
- Lambda calculus
- Partial recursive functions
- Tag systems (FSM plus FIFO queue)
- Unrestricted grammars:

  $a S a \rightarrow B$
- Post production systems
- Markov algorithms
- Conway's Game of Life
- One dimensional cellular automata
- DNA-based computing
- Lindenmayer systems

The Lambda Calculus

The successor function:

(λ *x*. *x* + 1) 3 = 4

Addition: *(λ x. λ y. x + y)* 3 4

This expression is evaluated by binding 3 to *x* to create the new function (*λ y.* 3 + *y*), which is applied to 4 to return 7.

In the pure lambda calculus, there is no built in number data type. All expressions are functions. But the natural numbers can be defined as lambda calculus functions. So the lambda calculus can effectively describe numeric functions.

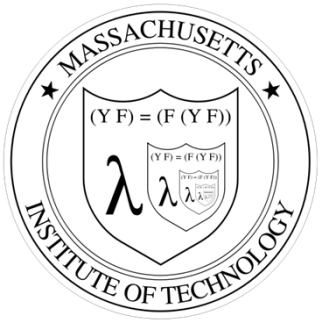
The Lambda Calculus

```
> (define Y
    (lambda (f)
      ((lambda (x) (f (lambda (y) ((x x) y))))
       (lambda (x) (f (lambda (y) ((x x) y)))))))
> (define H
    (lambda (g)
      (lambda (n)
        (if (zero? n)
            1
            (* n (g (- n 1)))))))
> ((Y H) 5)
120
>
```

The Lambda Calculus

The Applicative Y Combinator

```
> (((lambda (f)
    ((lambda (x) (f (lambda (y) ((x x) y))))
     (lambda (x) (f (lambda (y) ((x x) y))))))
  (lambda (g)
    (lambda (n)
      (if (zero? n)
          1
          (* n (g (- n 1)))))))
 5)
120
```



Tag Systems

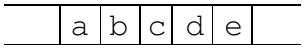
A tag system (or a Post machine) is an FSM augmented with a FIFO queue.

Simple for WW:
Not so simple for PalEven

The Power of Tag Systems

Tag systems are equivalent in power to Turing machines because the TM's tape can be simulated with the FIFO queue.

Suppose that we push `abcde` onto the queue:



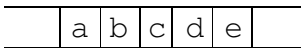
To read the queue, we must remove the `a` first.

But suppose we want to remove `e` first:

The Power of Tag Systems

Tag systems are equivalent in power to Turing machines because the TM's tape can be simulated with the FIFO queue.

Suppose that we push `abcde` onto the queue:



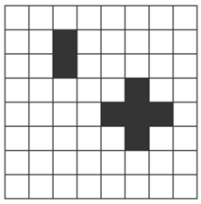
To read the queue, we must remove the `a` first.

But suppose we want to remove `e` first:

Treat the queue as a loop.

The Game of Life

Playing the game

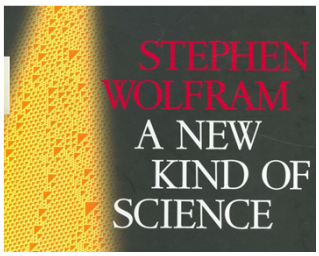


At each step of the computation, the value for each cell is determined by computing the number of neighbors (up to a max of 8) it currently has, according to the following rules:

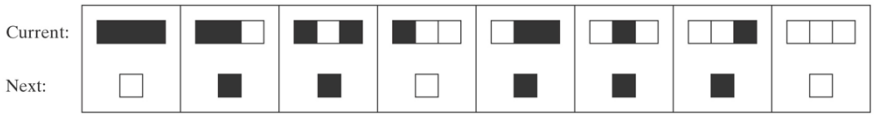
- A dead cell with exactly three live neighbors becomes a live cell (birth).
- A live cell with two or three live neighbors stays alive (survival).
- In all other cases, a cell dies or remains dead (overcrowding or loneliness).

We'll say that a game halts iff it reaches some stable configuration.

Elementary Cellular Automata



Wolfram's Rule 110 is a universal computer, if you can figure out how to encode the program and the input in the initial configuration:



For some fascinating pictures, look up Rule 110.
Conjectured in 1985 to be Turing complete, proved in 2000 by Matthew Cook.