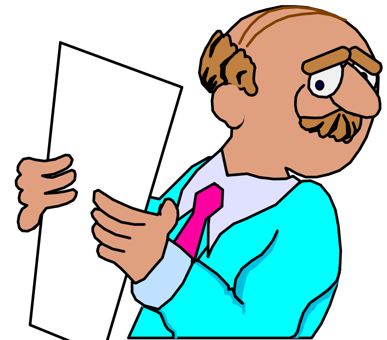
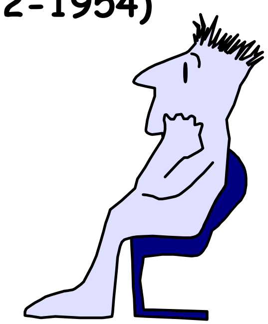
FSMs and Turing Machines

- Ways we know to compute
  - Truth-tables = combinational logic
  - State-transition tables = sequential logic
- Enumerating FSMs
- An even more powerful model: a "Turing Machine"
- What does it mean to compute?
- What CAN'T be computed
- Universal TMs = programmable TM

Alan Turing (1912-1954)



Church (1903-1995) Turing's PhD Advisor

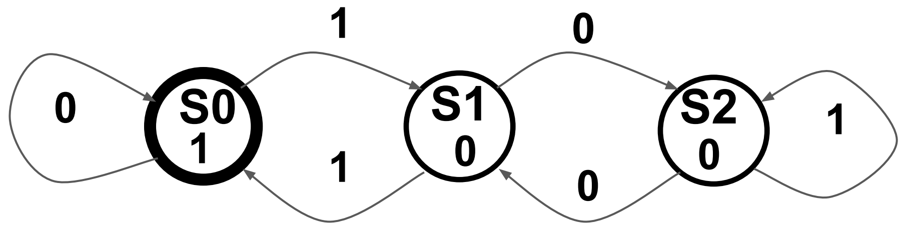


Post (1897-1954)

Kleene (1909-1994)

Let's play State Machine

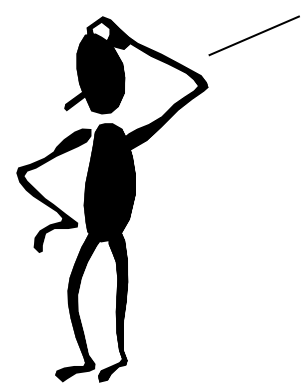
Let's emulate the behavior specified by the state machine shown below when processing the following string from LSB to MSB.



$$39_{10} = 0100111_2$$

input order

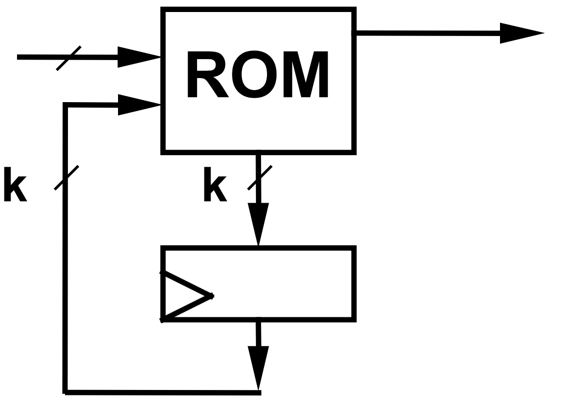
	State	Input	Next	Output
T=0	S0	1	S1	0
T=1	S1	1	S0	1
T=2	S0	1	S1	0
T=3	S1	0	S2	0
T=4	S2	0	S1	0
T=5	S1	1	S0	1
T=6	S0	0	S0	1



It looks to me like this machine outputs a 1 whenever the bit sequence that it has seen thus far is a multiple of 3. (Wow, and FSM can divide by 3!)

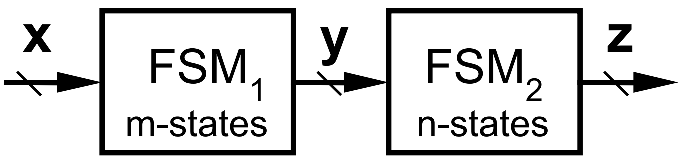
FSM Party Games

1. What can you say about the number of states?



States ≤ 2^k

2. Same question:



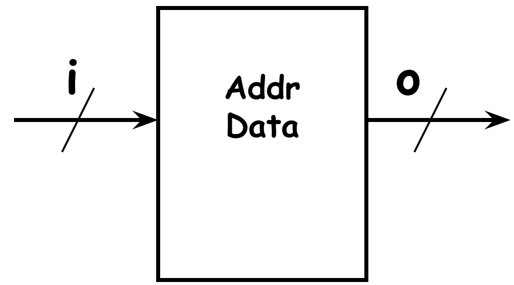
States ≤ m × n

2-Types of Processing Elements

Combinational Logic:

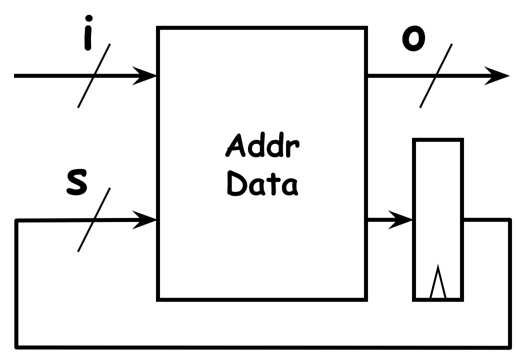
Table look-up, ROM

Recall that there are precisely 2^{2^i}, i-input combinational functions. A single ROM can store 'o' of them.



Finite State Machines:

ROM with State Memory



Thus far, we know of nothing more powerful than an FSM

Fundamentally, everything that we've learned so far can be done with a ROM and registers

FSMs as Programmable Machines

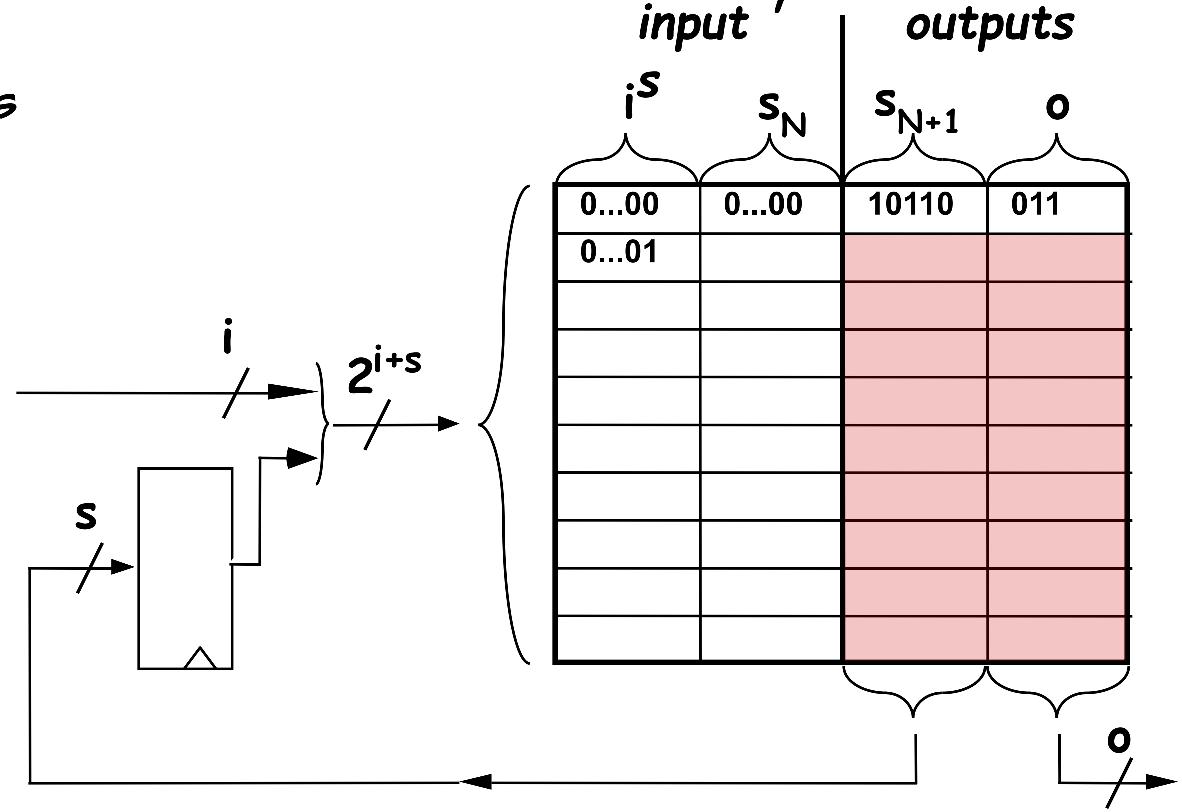
ROM-based FSM sketch:

Given i, s, and o, we need a ROM organized as:

2^{i+s} words x (o+s) bits

So how many possible
i-input,
o-output,
FSMs with
s-state bits
exist?

An FSM's behavior is completely determined by its ROM contents.



The number of "bits" in the ROM

All possible settings of the ROM's contents to: 1 or 0

$$2^{\overbrace{(o+s)2^{i+s}}}$$

(some may be equivalent)

How many state machines are there with 1-input, 1-output, and 1 state bit?

$$2^{(1+1)4} = 2^8 = 256$$

Recall how we were able to "enumerate" or "name" every 2-input gate? Can we do the same for FSMs?

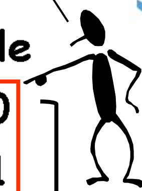
FSM Enumeration

GOAL: List all possible FSMs in some canonical order.

- INFINITE list, but
- Every FSM has an entry in and an associated index.

input		outputs	
i^s	s_N	o	s_{N+1}
0...00	0...00	10110	011
0...01			

These are the FSMs with 1 input and 1 output and 1 state bit. They have 8-bits in their ROM.

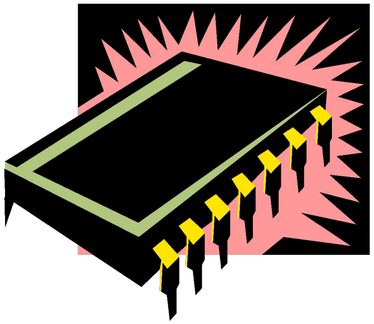


i	s	o	FSM#	Truth Table	
1	1	1	1	00000000	2^8 FSMs
1	1	1	2	00000001	
			...	...	
1	1	1	256	11111111	
2	2	2	257	000000...000000	2^{64}
2	2	2	258	000000...000001	
			18,446,744,073,709,551,872 ...	...	
3	3	3		000000...000000	
			3.9402 x 10^{115}...		
4	4	4		000000...000000	

Every possible FSM can be associated with a unique number. This requires a few wasteful simplifications. First, given an i-input, s-state-bit, and o-output FSM, we'll replace it with its equivalent n-input, n-state-bit and n-output FSM, where n is the greatest of i, s, and o. We can always ignore the extra input-bits, and set the extra output bits to 0. This allows us to discuss the i^{th} FSM

Some Favorites

FSM_{837}	modulo 3 state machine
FSM_{1077}	4-bit counter
FSM_{1537}	Combination lock
FSM_{89143}	Cheap digital watch
$FSM_{22698469884}$	RISC-V processor
$FSM_{23892749274}$	ARM7 processor
$FSM_{78436378389}$	Intel I-7 processor (Skylake)
$FSM_{78436378390}$	Intel I-7 processor (Kaby lake)



Can FSMs Compute Every Binary Function?

Nope!

There exist many simple problems that cannot be computed by FSMs.

For instance:

Checking for balanced parentheses

(()((()())) - Okay

(()())) - No good!

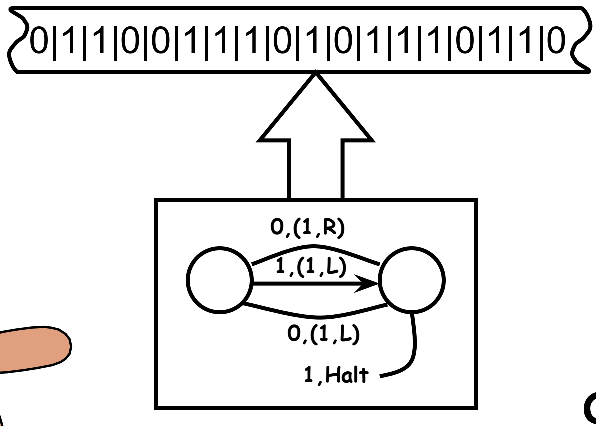
A function is specified by a deterministic output relationship for any given series of inputs, starting from a known initial state.

PROBLEM: Requires ARBITRARILY many states, depending on input.
Must "COUNT" unmatched LEFT parens.

But, an FSM can only keep track of a "bounded" number of events.
(Bounded by its number of states)

Is there another form of logic that can solve this problem?

Unbounded-Space Computation



DURING 1920s & 1930s, much of the "science" part of computer science was being developed (long before actual electronic computers existed). Many different "Models of Computation" were proposed, and the classes of "functions" that each could compute were analyzed.

One of these models was the "TURING MACHINE", named after Alan Turing (1912-1954).

Alan Turing

A Turing Machine is just an FSM which receives its inputs and writes outputs onto an "infinite tape". This simple addition overcomes the FSM's limitation that it can only keep track of a "bounded number of events".

A Turing Machine Example

Turing Machine Specification

- Infinite tape
- Discrete symbol positions
- Finite alphabet - say {0, 1}
- Control FSM

INPUTS:

Current symbol on tape

OUTPUTS:

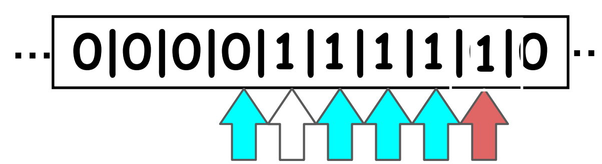
write 0/1

move tape Left or Right

- Initial Starting State {S0}
- Halt State {Halt}

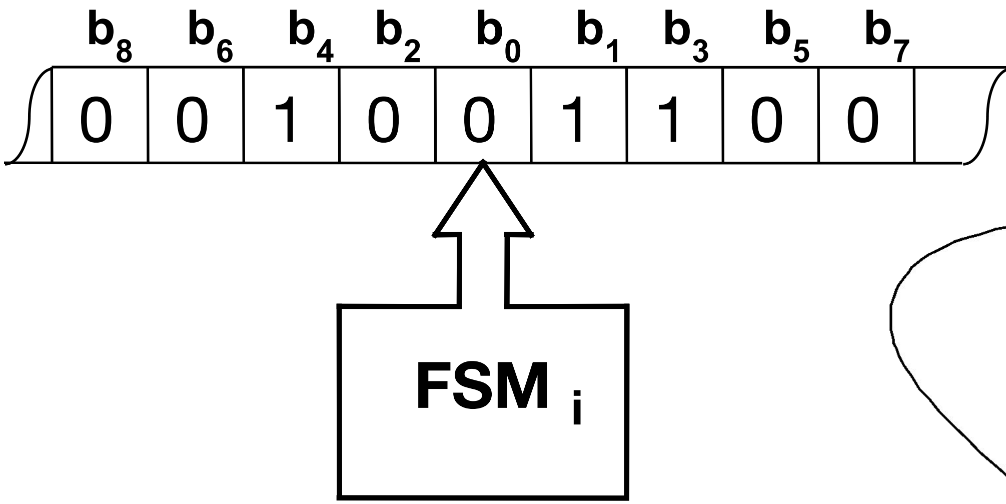
A Turing machine, like an FSM, can be specified via a state-transition table. The following Turing Machine implements a unary (base 1) counter.

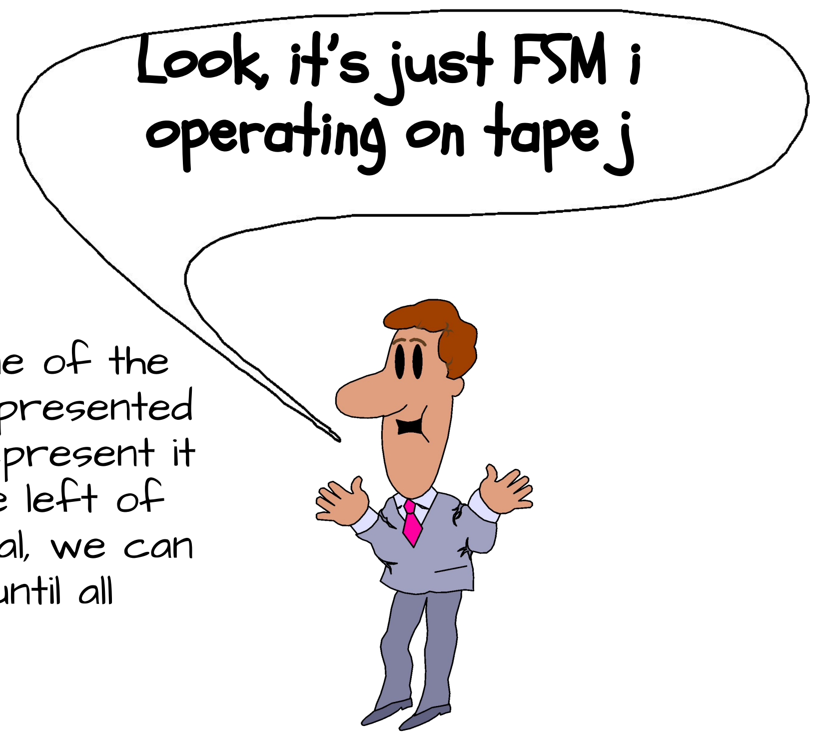
Current State	Tape Input	Write Tape	Move	Next State
S0	1	1	R	S0
S0	0	1	L	S1
S1	1	1	L	S1
S1	0	0	R	Halt



Turing Machine Tapes as Integers

Canonical names for bounded tape configurations:





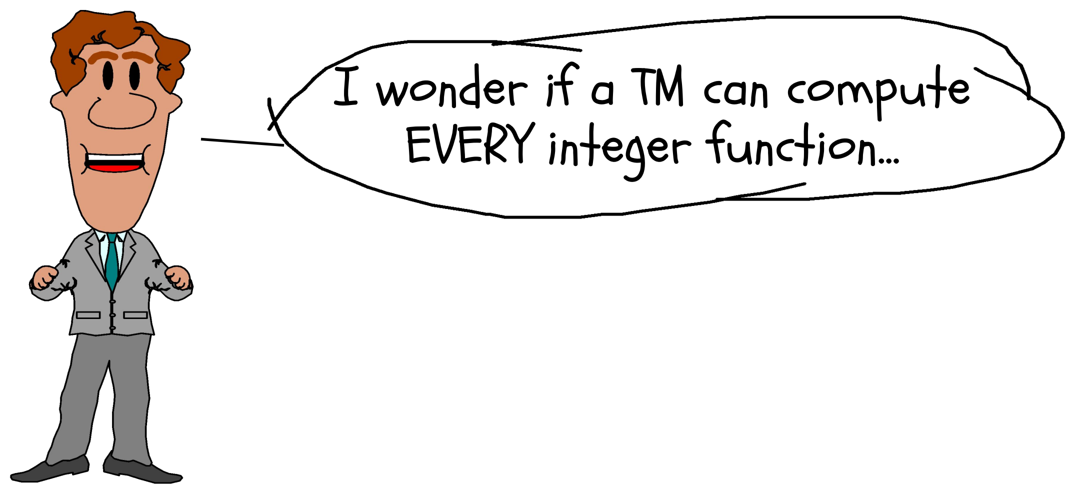
Note: The FSM part of a Turing Machine is just one of the FSMs in our enumeration. The tape can also be represented as an integer, but this is trickier. It is natural to represent it as a binary fraction, with a binary point just to the left of the starting position. If the binary number is rational, we can alternate bits from each side of the binary point until all that is left is zeros, then we have an integer.

TMs as Integer Functions

Turing Machine T_i operating on Tape x,
where $x = \ldots b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

$$y = T_i[x]$$

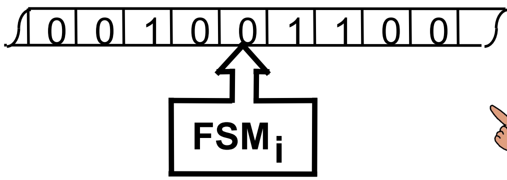
x: input tape configuration
y: output tape when TM *halts*



Alternative Models of Computation

Turing Machines [Turing]

Hardware head



Turing

Recursive Functions [Kleene]

$F(0,x) = x$

$F(y,0) = y$

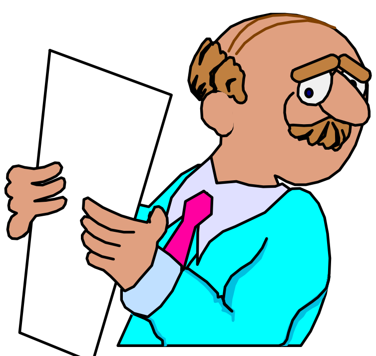
$F(y,x) = x + y + F(y-1,x-1)$

```
(define (fact n)
   (... (fact (- n 1)) ...)
```

Theory head

Kleene (1909-1994)

Lambda calculus [Church, Curry, Rosser...]



Math head

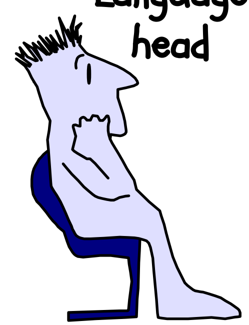
$\lambda x.\lambda y.xxy$

```
(lambda(x)(lambda(y)(x (x y))))
```

Church (1903-1995)
Turing's PhD Advisor

Production Systems [Post, Markov]

Language head



$\$_0 \rightarrow [\,]$

$\$ \rightarrow [\$]$

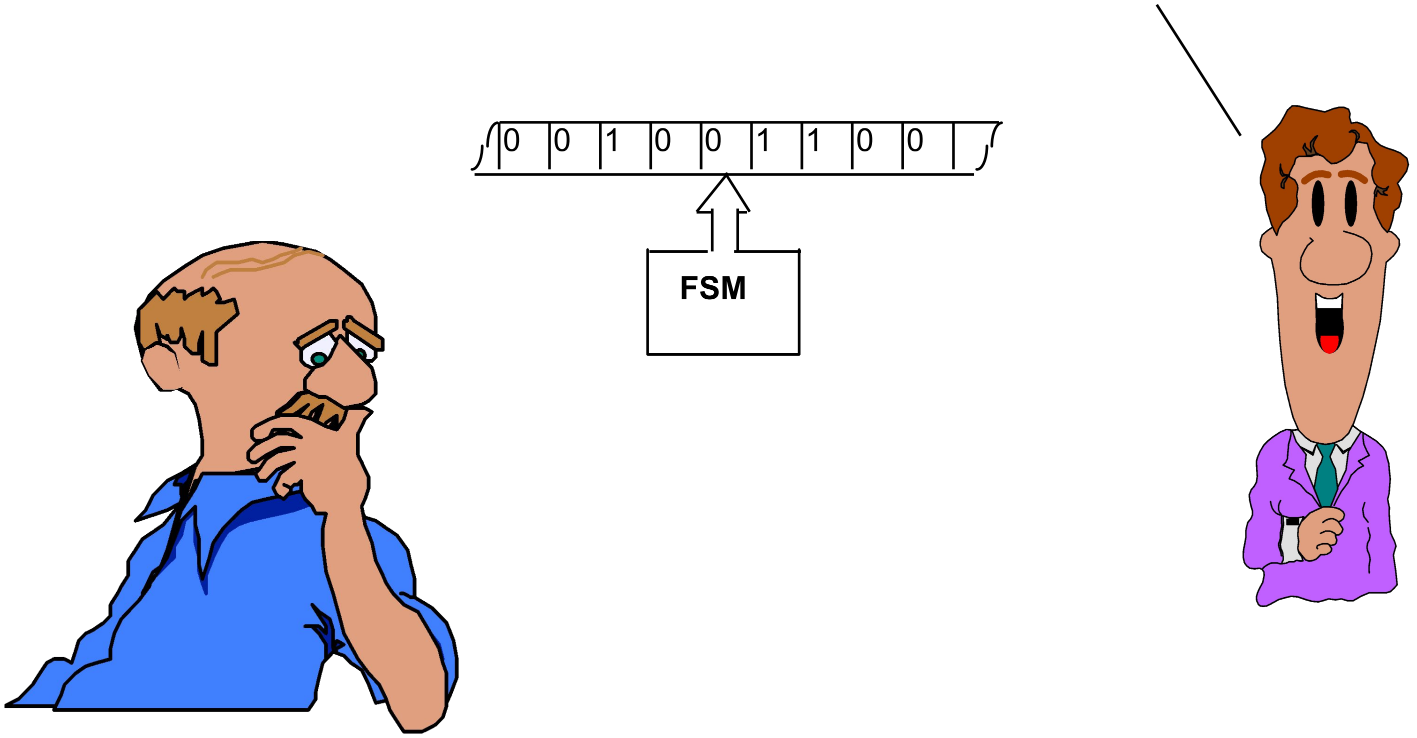
$\$ \rightarrow \$\$$

$\$_i [\,] \$_j \rightarrow \$_i \$_j$

Post
(1897-1954)

The 1st Computer Industry Shakeout

Here's a TM that
computes SQUARE ROOT!



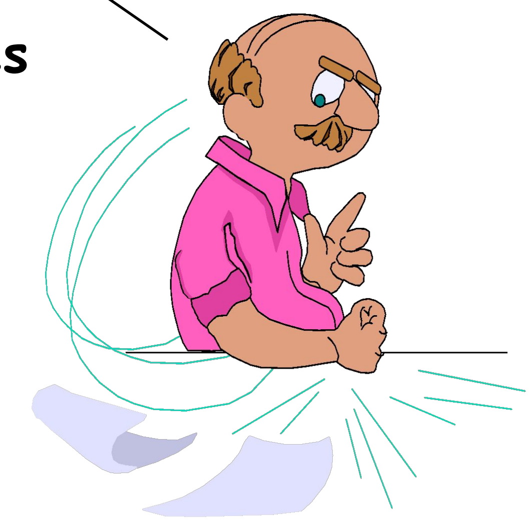
And the Battles Raged

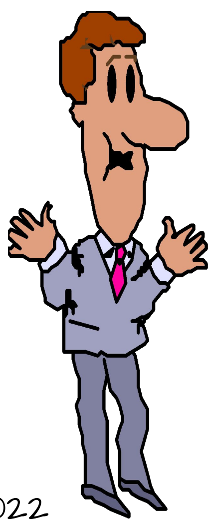
Here's a Lambda Expression
that does the same thing...

`(λ(x)  .....)`

... and here's one that computes
the n^{th} root for ANY n!

`(λ(x n)  .....)`





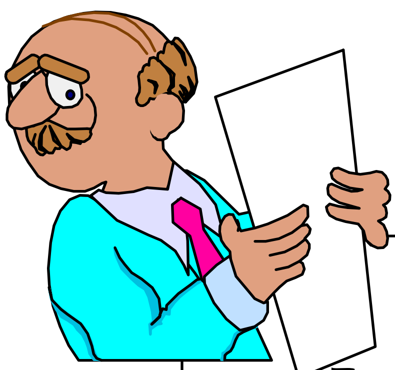
A Fundamental Result

Turing's amazing proof: Each model is capable of computing <u>exactly</u> the same set of integer functions! None is more powerful than the others.

Proof Technique: Constructions that translate between models

BIG IDEA: Computability, independent of computation scheme chosen

This means that we know of no mechanisms (including computers) that are more "powerful" than a Turing Machine, in terms of the functions they can compute.



Church's Thesis:

Every discrete function computable by ANY realizable machine is computable by some Turing machine.

Computable Functions

The "input" to our computable function will be given on the initial tape, and the "output" will be the contents of the tape when the TM halts.

$f(x)$ *computable* $<=>$ for some k, all x:

$$f(x) = T_K[x] \equiv f_K(x)$$

Representation tricks: to compute $f_k(x,y)$ (2 inputs)

$<x,y> \equiv$ integer whose *even* bits come from x,
and whose *odd* bits come from y; whence

$$f_K(x, y) \equiv T_K[<x, y>]$$

$f_{12345}(x,y) = x * y$

$f_{23456}(x) = 1$ iff x is prime, else 0

TMs, Like Programs, Can Misbehave

It is possible that a given Turing Machine may not produce a result for a given input tape. And it may do so by entering an infinite loop!

Consider the given TM.

It scans a tape looking for the first non-zero cell to the right.

What does it do when given a tape that has no 1's to its left?

We say this TM does not halt for that input!

Current State	Tape Input	Write Tape	Move	Next State
S0	1	1	L	Halt
S0	0	0	R	S0

$tape_{256}$ = … 0|0|0|0|0|0|0|1|0|0 …

$tape_{8}$ = … 0|1|0|0|0|0|0|0|0|0 …

Enumeration of Computable functions

Conceptual table of TM behaviors…

VERTICAL AXIS: Enumeration of TMs.

HORIZONTAL AXIS: Enumeration of input tapes.

(j ,k) entry = result of $TM_k[j]$ -- integer, or * if it never halts.

Turing Machine Tapes →

Turing Machine FSMs ↓

	$f_i(0)$	$f_i(1)$	$f_i(2)$	…	$f_i(j)$	…
f_0	~~37~~ 1	~~23~~ 1	~~*~~ 0	…	…	
f_1	~~42~~ 1	~~*~~ 0	~~666~~ 1	…	…	
…	…	…	…	…	…	
f_k	…	…	…	…	$f_k(j)$	
…						

Every computable function is in this table, since everything that we know how to compute can be computed by a TM.

Do there exist well-specified integer functions that a TM can't compute?

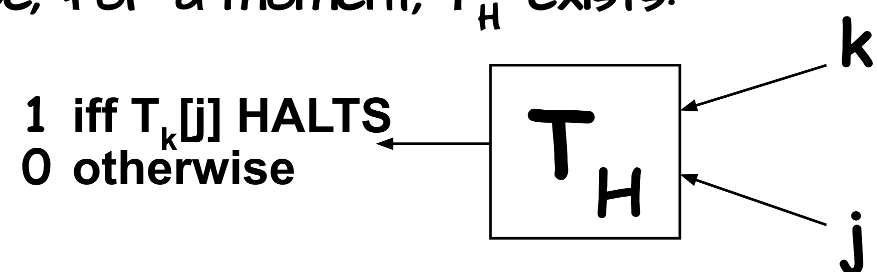
The Halting Problem: Given j, k: Does TM_k Halt with input j?

The Halting Problem

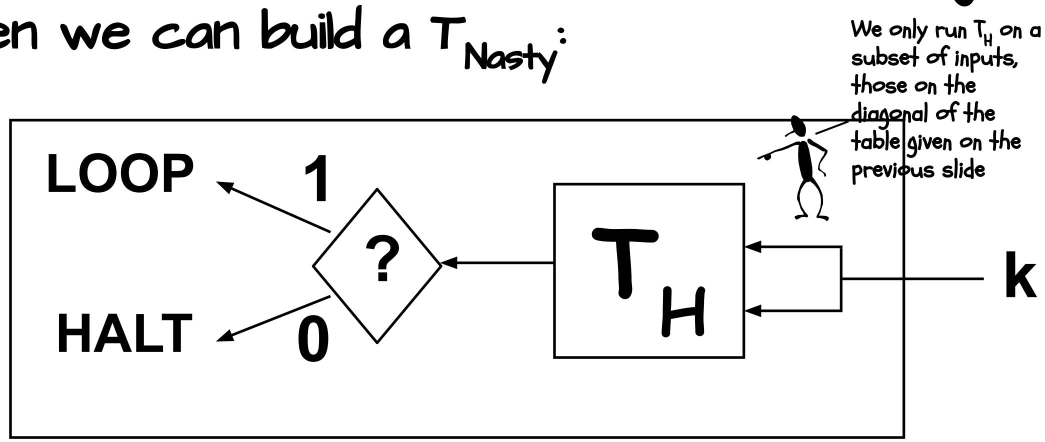
The Halting Function: $T_H[k, j] = 1$ iff $TM_k[j]$ halts, else 0

Can a Turing machine compute this function?

Suppose, for a moment, T_H exists:



Then we can build a T_{Nasty}:



$T_{Nasty}[k]$ LOOP if $T_k[k] = 1$ (halts)
HALT if $T_k[k] = 0$ (loops)

What does T_{Nasty}[Nasty] do?

Answer:

T_{Nasty}[Nasty] loops if T_{Nasty}[Nasty] halts
T_{Nasty}[Nasty] halts if T_{Nasty}[Nasty] loops

That's a contradiction.

Thus, T_H is not computable by a Turing Machine!

<u>Net Result</u>: There are some integer functions that Turing Machines simply cannot answer. Since, we know of no better model of computation than a Turing machine, this implies that there are some well-specified problems that defy computation.

Limits of Turing Machines

A Turing machine is formal abstraction that addresses

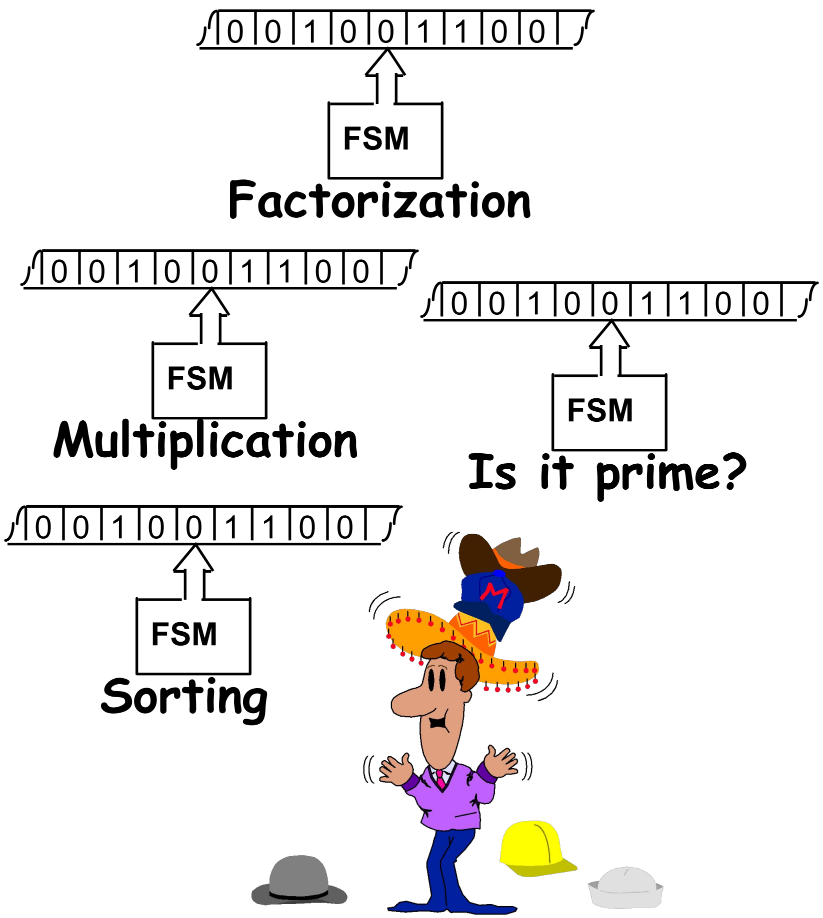
- Fundamental Limits of Computability -
  - What is means to compute.
  - The existence of uncomputable functions.
- We know of no machine more powerful than a Turing machine in terms of the functions that it can compute.

But they ignore

- Practical coding of programs
- Performance
- Implementability
- Programmability

… these latter issues are the primary focus of contemporary computer science (Remainder of Comp 411)

Computability vs. Programmability



Recall Church's thesis:

"Any discrete function computable by ANY realizable machine is computable by some Turing Machine"

We've defined what it means to COMPUTE (whatever a TM can compute), but, a Turing machine is nothing more that an FSM that receives inputs from, and outputs onto, an infinite tape.

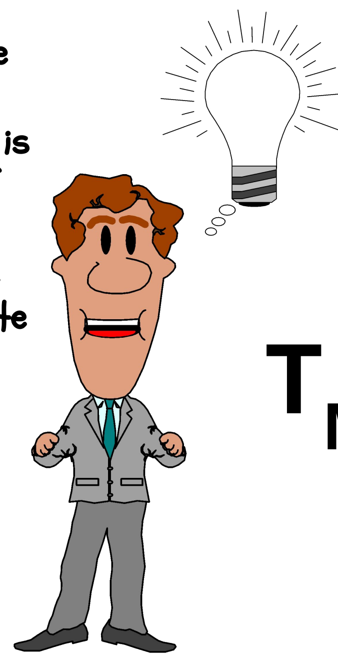
So far, we've been designing **a new FSM for each new Turing machine** that we encounter.

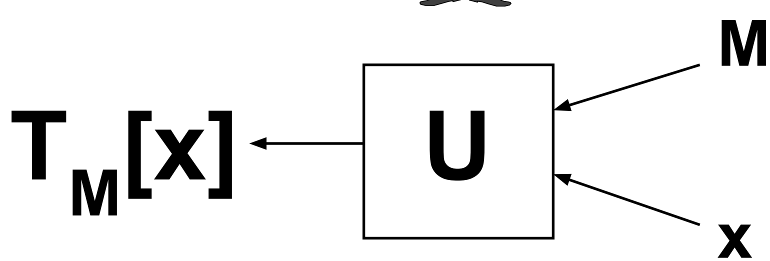
Wouldn't it be nice if we could design a more general-purpose Turing machine?

Programs as Data

What if we encoded the description of the FSM on our tape, and then wrote a general purpose FSM to read the tape and *EMULATE* the behavior of the encoded machine? We could just store the state-transition table for our TM on the tape and then design a new TM that makes reference to it as often as it likes. It seems possible that such a machine could be built.

"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine M, then U will compute the same sequence as M."
– Turing 1936 (Proc of the London Mathematical Society, Ser. 2, Vol. 42)





Fundamental Result: Universality

Define "Universal Function": $U(x,y) = T_x(y)$ for every x, y ...
Surprise! **U(x,y) IS COMPUTABLE**,
hence $U(x,y) = T_U(\langle x,y \rangle)$ for some U.

Universal Turing Machine (UTM):

$$T_U[\langle y, z \rangle] = T_y[z]$$

tape = "data"
TM = "program"
"interpreter"

PARADIGM for General-Purpose Computer!

INFINITELY many UTMs ...
Any one of them can evaluate any computable function by simulating/emulating/interpreting the actions of Turing machine given to it as an input.

UNIVERSALITY:
Basic requirement for a general purpose computer

Demonstrating Universality

Suppose you've designed Turing Machine T_k and want to show that its universal.

APPROACH:

1. Find some known universal machine, say T_u.
2. Devise a program, P, to *simulate* T_u on T_k:
   $T_k[\langle P,x\rangle] = T_u[x]$ for all x.
3. Since $T_u[\langle y,z\rangle] = T_y[z]$, it follows that, for all y and z.

Turing Complete

$$T_K[\langle P, \langle y,z\rangle\rangle] = T_U[\langle y,z\rangle] = T_y[z]$$

CONCLUSION: Armed with program **P**, machine $\mathbf{T_K}$ can mimic the behavior of an arbitrary machine $\mathbf{T_y}$ operating on an arbitrary input tape z.

HENCE $\mathbf{T_K}$ can compute any function that can be computed by any Turing Machine.

Next Time

Enough theory already, let's build something!