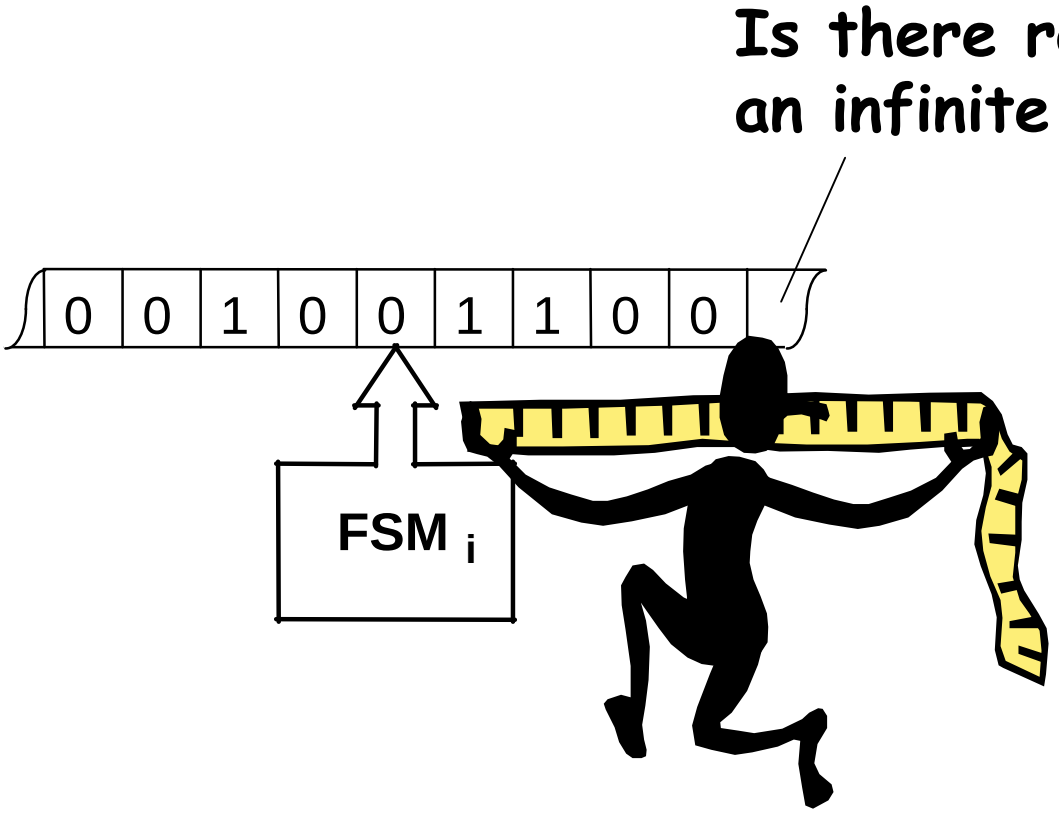
Models of Computation: Programmability



Handouts: Lecture Notes, PS5

3-Types of Processing Elements

Combinational Logic:

Table look-up, ROM

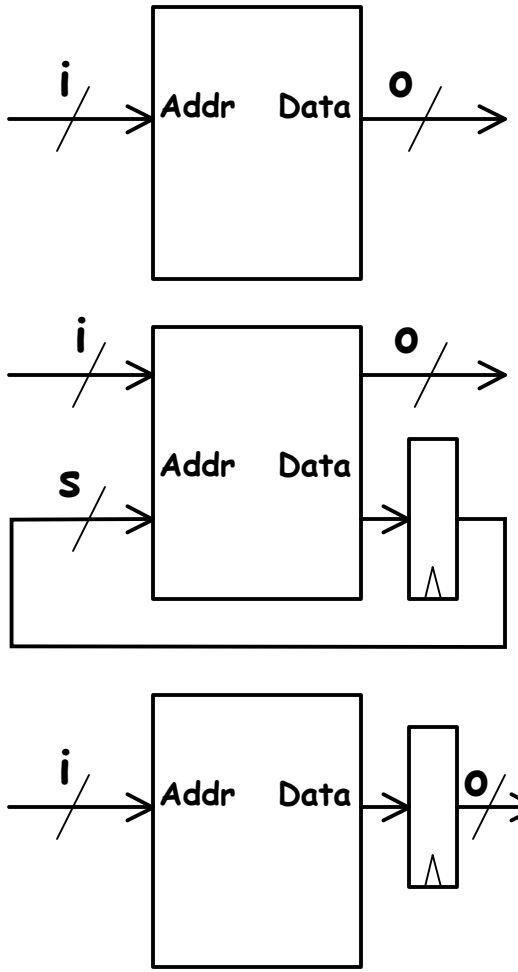
Finite State Machines:

ROM with Feedback

Pipelined Processing:

ROM with storage for intermediate results

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:

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

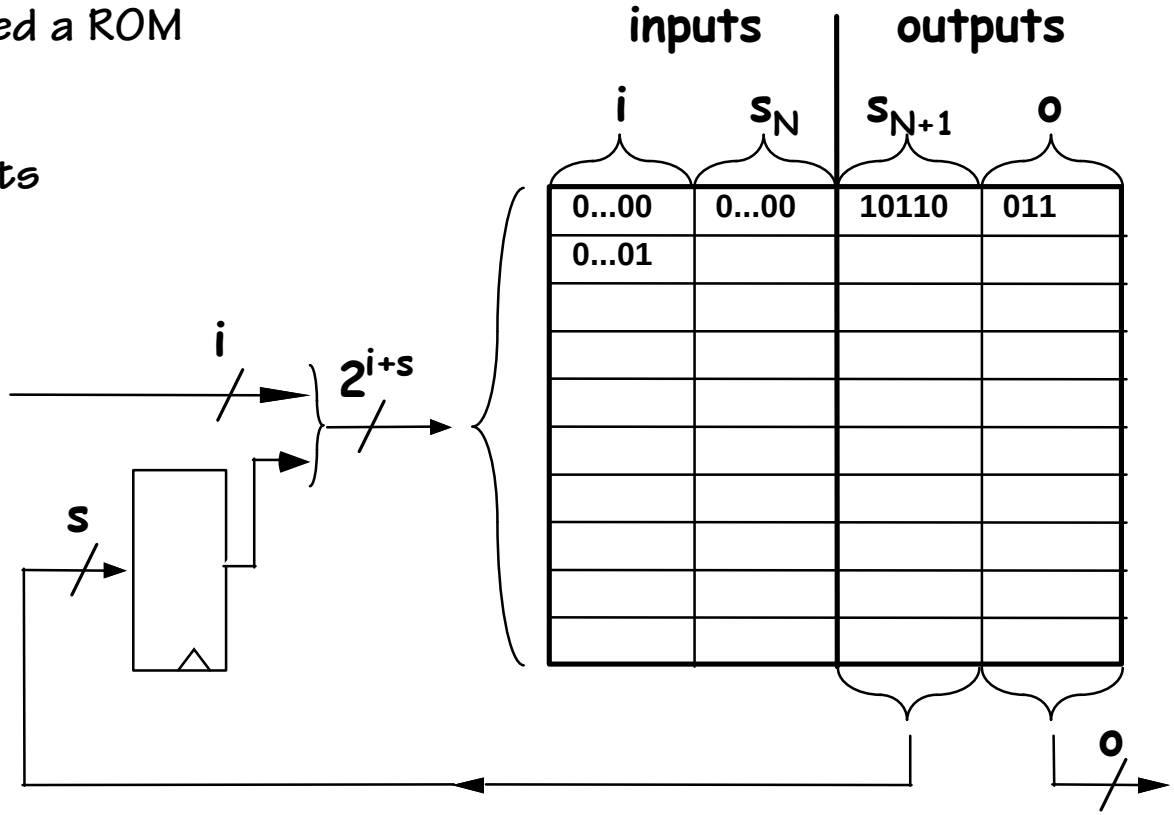
Given i, s, and o,we need a ROM organized as:

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

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

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

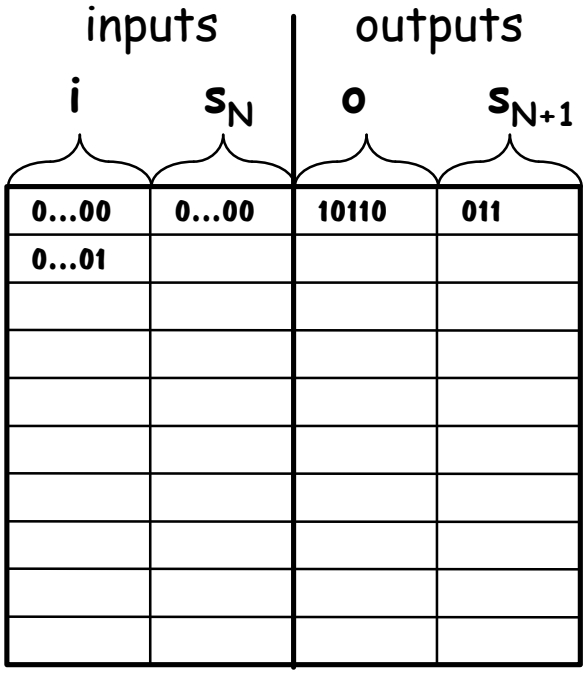
(some may be equivalent)



FSM Enumeration

GOAL: List all possible FSMs in some canonical order.

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

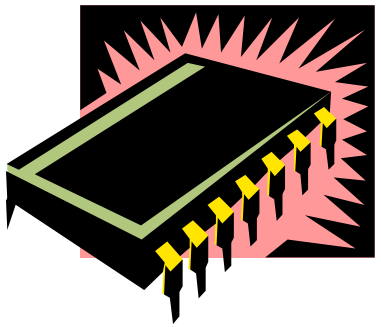


i	s	o	FSM#	Truth Table	
1	1	1	1	00000000	2^8 FSMs
1	1	1	2	00000001	
			…	…	
1	1	1	256	11111111	
2	2	2	257	000000…000000	2^{64}
2	2	2	258	000000…000001	
			…	…	
3	3	3		000000…000000	
			…		
4	4	4		000000…000000	

Every possible FSM can be associated with a number. We can discuss the i^{th} FSM

Some Perennial Favorites...

FSM_{837}	modulo 3 counter
FSM_{1077}	4-bit counter
FSM_{1537}	lock for 6.004 Lab
FSM_{89143}	Cheap digital watch
$FSM_{22698469884}$	Intel Pentium CPU – rev 1
$FSM_{784362783}$	Intel Pentium CPU – rev 2
$FSM_{784363783}$	Intel Pentium II CPU



Are FSMs the ULTIMATE computation device?

There exist common problems that cannot be computed by FSMs. For instance:

Checking for balanced parenthesis

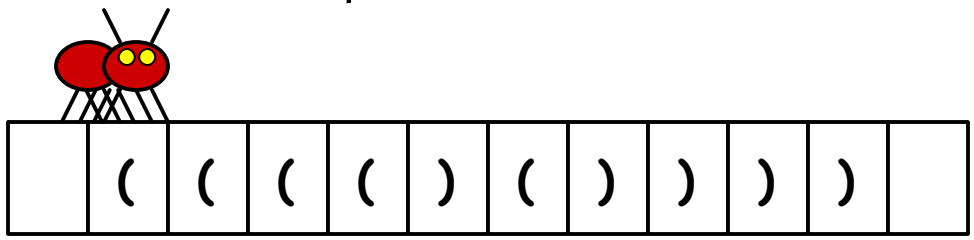
(()(()())) - Okay
(()())) - No good!

PROBLEM: Requires ARBITRARILY many states, depending on input. Must "COUNT" unmatched LEFT parens. **An FSM can only keep track of a finite number of objects.**

Do we know of a machine that can solve this problem?

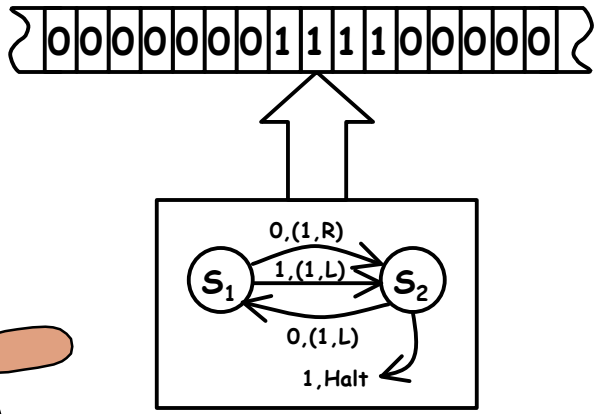
Yes, Roboant can!

State	Input	Crumb?	Crumb	Move	Next State	Comment
S1	(	-	Y	R	S1	Mark open paren
S1	)	-	Y	L	S2	Mark close paren
S1	sp	-	N	L	S4	Reached end
S2	-	N	N	L	S2	Scan back to last open
S2	-	Y	N	R	S3	Eat crumb
S3	-	N	N	R	S3	Goto close paren
S3	-	Y	N	R	S1	Eat crumb
S4	(or)	N	N	L	S4	Move Left
S4	(or)	Y	N	L	S5	Unmatched/Eat
S4	sp	-	Y	L	Halt	Matched
S5	(or)	-	N	L	S5	Unmatched/Eat
S5	sp	-	N	L	Halt	Unmatched



What is it that makes Roboant so powerful? RoboAnt is very FSM-like.
Is there exist some extension to an FSM that allows it to "compute" more?

Unbounded-Space Computation



Alan Turing

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" which could be computed by each were analyzed.

One of these models was the TURING MACHINE named after Alan Turing.

A Turing Machine is just an FSM which receives its inputs and writes outputs onto an infinite tape...

Solves "FINITE" problem of FSMs.

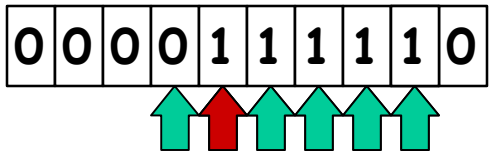
A Turing Machine Example

Turing Machine Specification

- Doubly-infinite tape
- Discrete symbol positions
- Finite alphabet – say {0, 1}
- Control FSM
  - INPUTS:
    - Current symbol
  - OUTPUTS:
    - write 0/1
    - move Left/Right
- Initial Starting State {S0}
- Halt State {Halt}

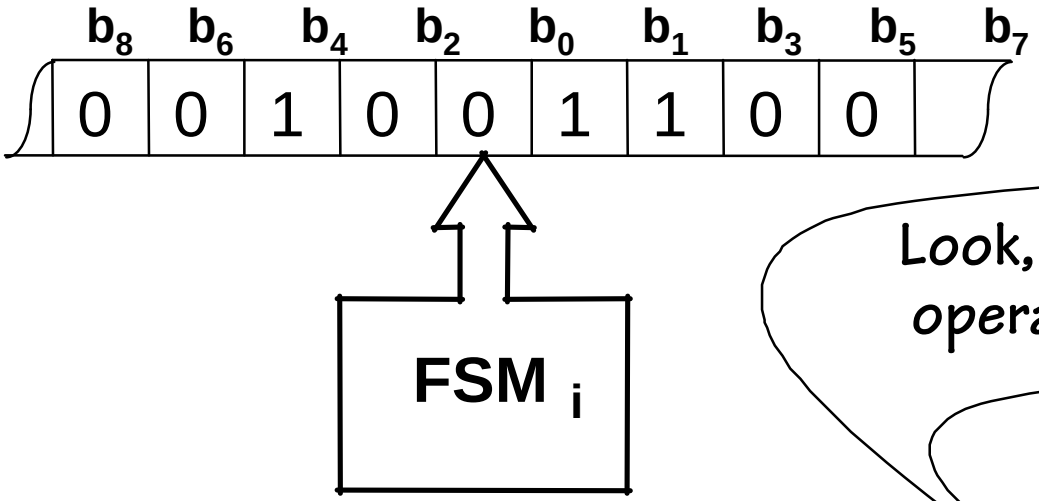
A Turing machine, like an FSM, can be specified with a truth table. The following Turing Machine implements a unary (base 1) incrementer.

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:

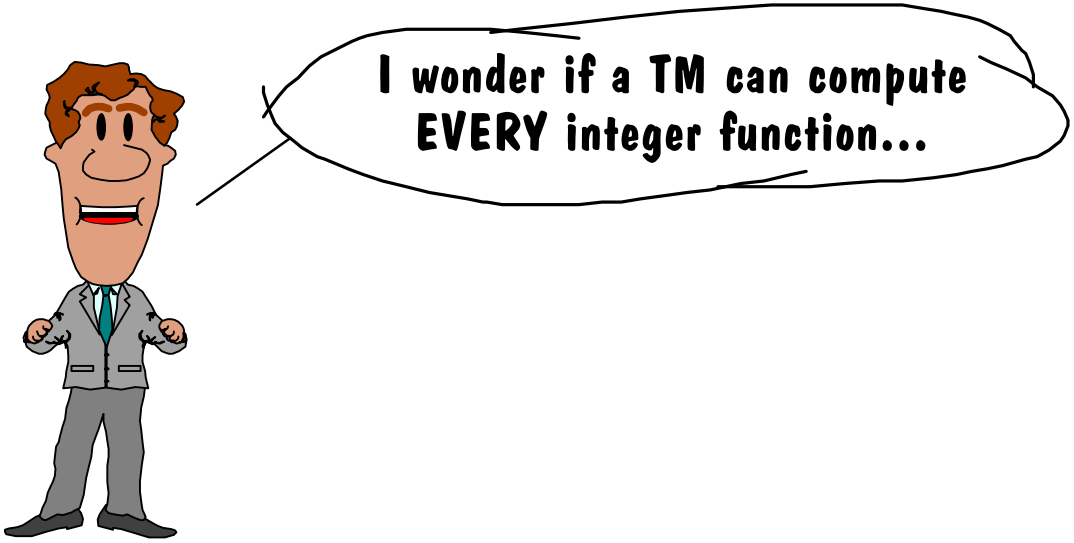


TMs as Integer Functions

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

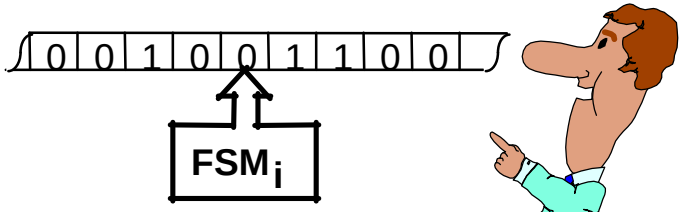
$$y = T_i[x]$$

x: input tape configuration
y: output tape configuration



Alternative Models of Computation

Turing Machines [Turing]



Turing

Recursive Functions [Kleene]

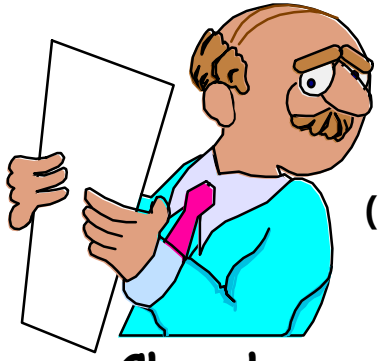
F(0,x) ? x

F(1+y,x) ? 1+F(x,y)

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

Kleene

Lambda calculus [Church, Curry, Rosser…]

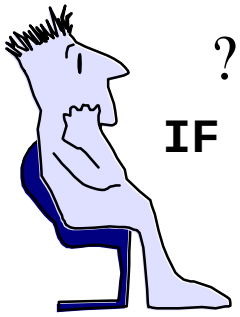


?x.?y.xxy

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

Church

Production Systems [Post, Markov]



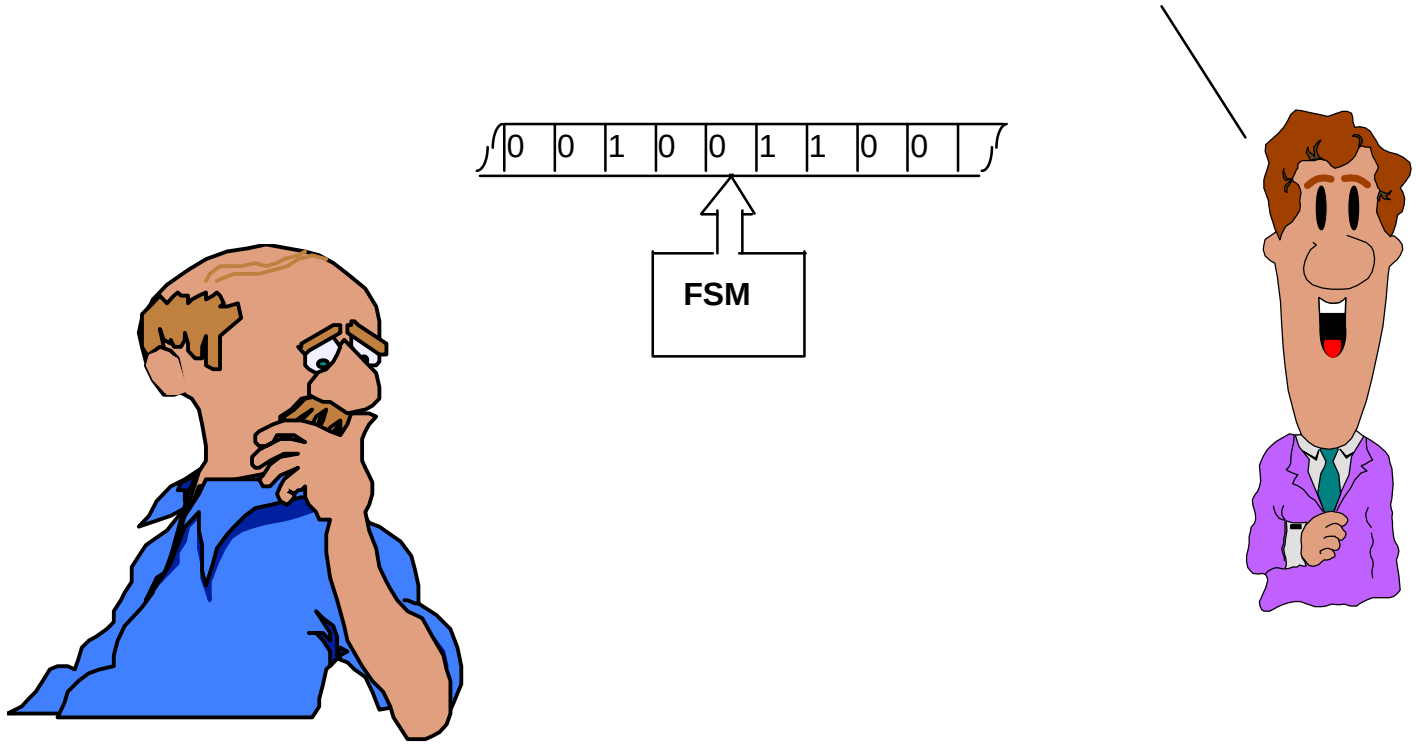
? ? ?

```
IF pulse=0 THEN
   patient=dead
```

Post

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...

```
(?(x) .....)
```

... and here's one that computes
the nth root for ANY n!

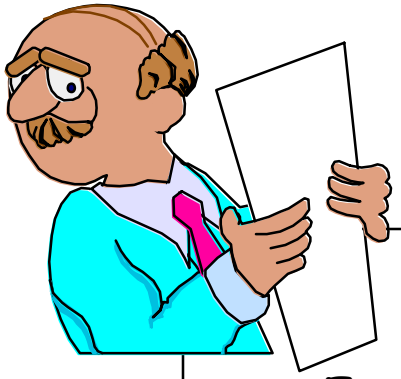
```
(?(x n) .....)
```

Fundamental Result #1: Computable Functions

Each model is capable of computing <u>exactly</u> the same set of integer functions!

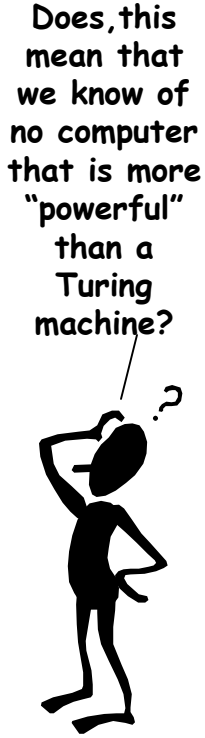
Proof Technique: Constructions that translate between models

BIG IDEA: Computability, independent of computation scheme chosen



<u>Church's Thesis:</u>

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



Computable Functions

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

$$f(x) = T_k[x] \equiv f_k(x)$$

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

<x,y> ??integer whose even bits come from x, and whose odd bits come from y; whence

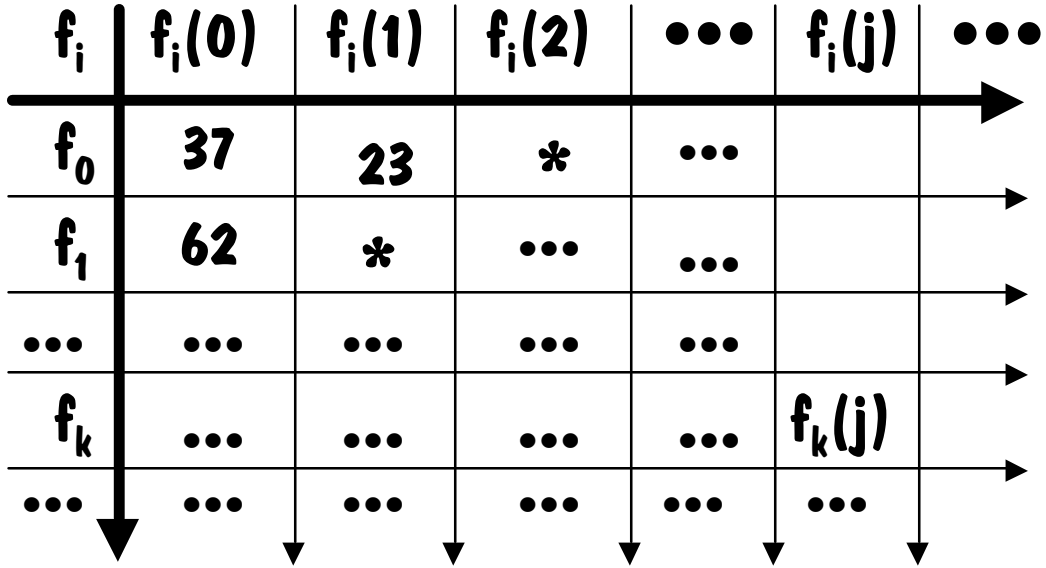
$$f_K(x, y) \equiv T_K[<x, y>]$$

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

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

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 never halts.



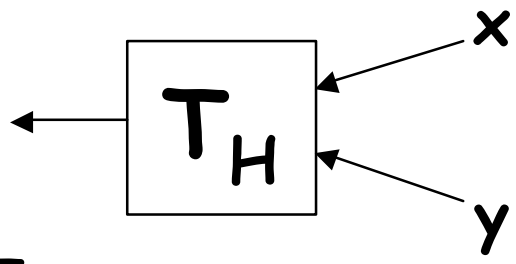
The Halting Problem: Given j, k: Does TM_k Halt with input j?

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?

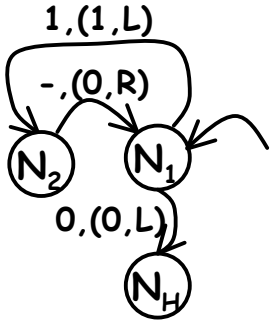
Suppose T_H exists:

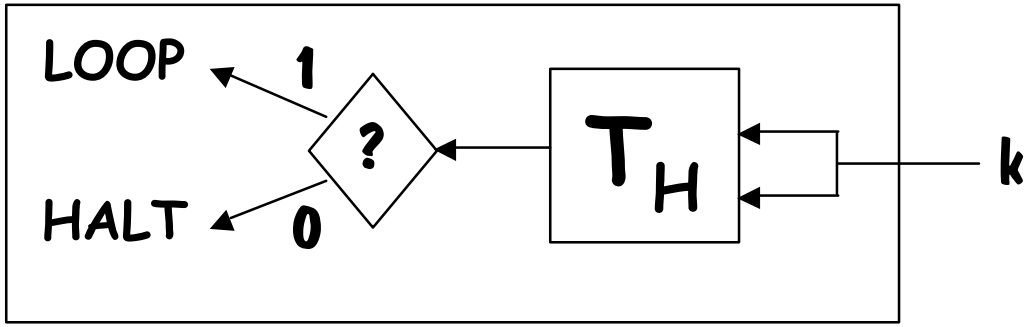


If T_H is computable then so is T_{Nasty}

Then we can build a T_{Nasty}:

Replace the Halt state of TH with this.





$T_{Nasty}[k]$ **LOOPS if** $T_k[k]$ halts
HALTS if $T_k[k]$ loops

What does T_{Nasty}[Nasty] do?

Answer:

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

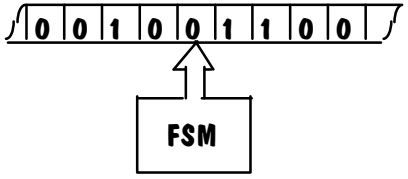
T_{Nasty}[Nasty] halts if T_{Nasty}[Nasty] loops

That's a contradiction.

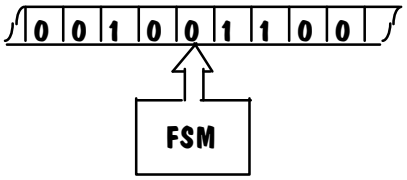
Thus, T_H is uncomputable by a Turing Machine!

There are some questions 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 questions that defy computation.

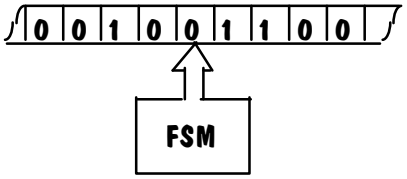
Too many Turing machines!



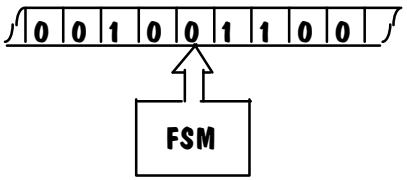
Multiplication



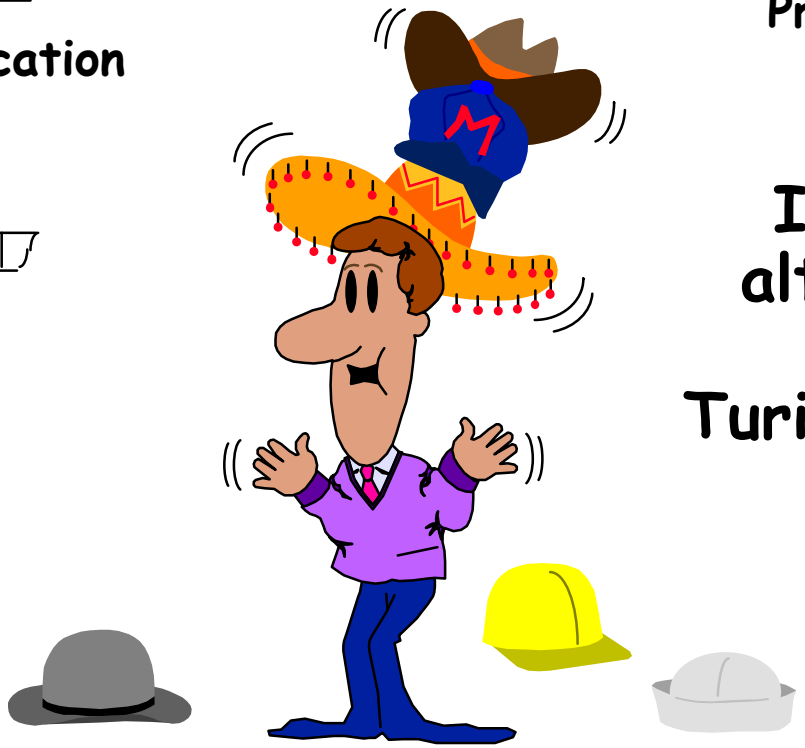
Factorization



Primality Test



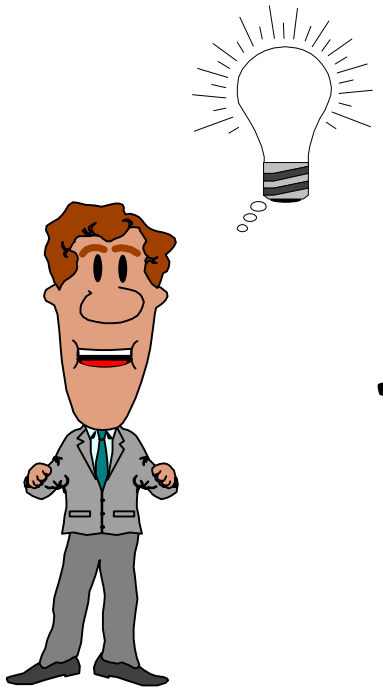
Sorting

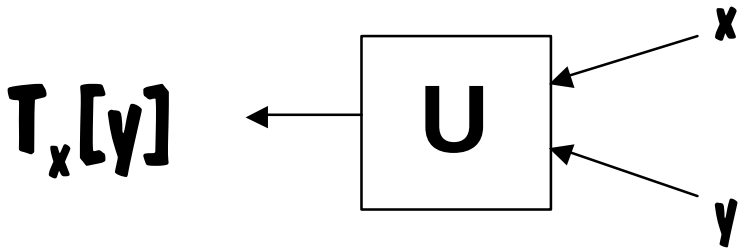


Is there an alternative to ad-hoc Turing Machines?

Program as "Input"

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? Since the FSM is just a look-up table, and our machine can make reference to it as often as it likes, it seems possible that such a machine could be built.



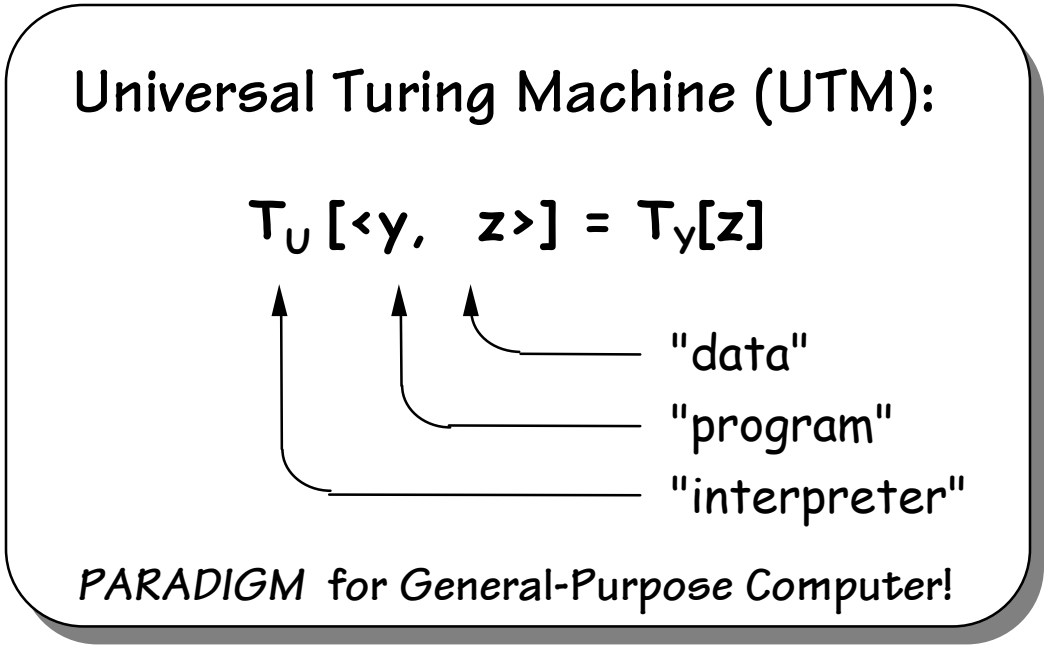


Fundamental Result #2: Universality

Define "Universal Function": $U(x,y) = T_x(y)$ for every x, y ...

Surprise! U is computable,

hence $U(x,y) = T_U(\langle x,y \rangle)$ for some U.



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:**
   $T_K[<P,x>] = T_U[x]$ for all x.
3. **Since $T_U[<y,z>] = T_Y[z]$, it follows that, for all y and z.**

$$T_K\,[<P,<y,z>>] \;=\; T_U[<y,z>] \;=\; T_y[z]$$

CONCLUSION: Armed with program **P**, machine T_K can mimic the behavior of an arbitrary machine T_Y operating on an arbitrary input tape z.

HENCE T_K can compute any function that can be computed by any Turing Machine.

Interpretive Layers: What's going on?

$$T_K\,[<P,<y,z>>] \;=\; T_U[<y,z>] \;=\; T_y[z]$$

Multiple levels of interpretation:

$T_y[z]$	Application (Desired user function)
$T_U[<y,z>]$	Portable Language / Virtual Machine
$T_K[<P,<y,z>>]$	Computing Hardware / Bare Metal

Benefits of Interpretation:

BOOTSTRAP high-level functionality on very simple hardware.

Deal with "***IDEAL***" machines rather than real machines.

REAL MACHINES are built this way - several interpretive layers.

Power of Interpretation

BIG IDEA: Manipulate *coded representations* of computing machines, rather than the machines themselves.

- PROGRAM as a behavioral description
- SOFTWARE vs. HARDWARE
- INTERPRETER as machine which takes program and mimics behavior it describes
- LANGUAGE as interface between interpreter and program
- COMPILER as translator between languages:

INTELLECTUAL BENEFITS:

- Programs as data -- mathematical objects
- Combination, composition, generation, parameterization, etc.

Reality: Limits of Turing Machines

These formal abstractions address

- Fundamental Limits of Computability
- Basic ideas: Interpretation, Algorithm

But they ignore

- Practical coding of programs
- Performance
- Implementability
- Programmability

... these latter issues are the primary focus of contemporary computer science (6.001, 6.004)

Next time: Designing an Instruction Set