State Machines

1) State Machine Design
2) How can we improve on FSMs?
3) Turing Machines
4) Computability

Example FSM: Roboant

SENSORS: antennae L and R, each 1 if in contact with something.

ACTUATORS: Forward Step F, ten-degree turns TL and TR (left, right).

GOAL: Make our ant smart enough to get out of a maze like:

STRATEGY: "Right antenna to the wall"
Lost in Space

Action: Go forward until we hit something.

Bonk!

Action: Turn left (CCW) until we don’t touch anymore
A Little to the Right...

Action: Step and turn right a little, look for wall

Then a Little to the Left

Action: Step and turn left a little, till not touching (again)
Dealing with Corners

Action: Step and turn right until we hit perpendicular wall

Equivalent State Reduction

Observation: $S_i \equiv S_j$ if
1. States have identical outputs; AND
2. Every input $\rightarrow$ equivalent states.

Reduction Strategy:
Find pairs of equivalent states, MERGE them.
An Evolutionary Step

*Merge* equivalent states Wall1 and Corner into a single new, combined state.

Behaves exactly as previous (5-state) FSM, but requires half the ROM in its implementation!

---

Building the Transition Table

Recall that a state Transition diagram specifies the function of a state machine, not its implementation. Next, we have to convert our specification to gates and registers. To do so, we rewrite our state transition diagram as a truth table. Each arc of the state transition diagram contributes one or more rows to our truth table.
### Implementation Details

Eventually, we’ll complete the entire truth table. At this point we can directly implement the state machine using a ROM and STATE REGISTERS. Alternately, we can build a minimal SOP realization using Karnaugh maps.

<table>
<thead>
<tr>
<th>S</th>
<th>L</th>
<th>R</th>
<th>S'</th>
<th>TR</th>
<th>TL</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
S_1' = S_1 S_0 + \overline{L} S_1 + \overline{L} R S_0
\]

<table>
<thead>
<tr>
<th>S</th>
<th>L</th>
<th>R</th>
<th>S'</th>
<th>TR</th>
<th>TL</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
S_0' = R + \overline{L S_1} + L S_0
\]

### Ant Schematic

Are you telling me that essence amounts to no more than 15 gates!
Super Roboant

Featuring the new Mark-II ant: It can add crumbs (M), erase crumbs (E), and sense (S) crumbs along its path.

Finite State Machines: An Important Abstraction

- **FORMAL MODEL:** Substantial literature, theoretical results.

- **PRACTICAL IMPACT:** Major engineering tool...

- **METAPHYSICAL IMPACT:** Are we all just complicated FSMs? Perhaps with imperfect state registers and “fuzzy” logic...

... You'll see FSMs for the rest of your life!
## 2-Flavors of Processing Elements

**Combinational Logic:**
- Table look-up, ROM

**Finite State Machines:**
- ROM with Feedback

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

### FSMs as Programmable Machines

**ROM-based FSM sketch:**
- Given `i`, `s`, and `o`, we need a ROM organized as:
  \[2^i \times (o+s)\text{ bits}\]
- So how many possible `i`-input, `o`-output, FSMs with `s`-state bits exist?
  \[2^{(o+s)2^{i+s}}\]

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

**GOAL:** List all possible FSMs in some canonical order.
- INFINITE list, but
- Every FSM has an entry in and an associated index.

<table>
<thead>
<tr>
<th>i</th>
<th>s</th>
<th>o</th>
<th>FSM#</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>00000001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>256</td>
<td>256</td>
<td>11111111</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>257</td>
<td>257</td>
<td>000000...000000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>258</td>
<td>258</td>
<td>000000...000001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>000000...000000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>000000...000000</td>
</tr>
</tbody>
</table>

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 ith FSM.

Some Perennial Favorites...

- FSM_{837} modulo 3 counter
- FSM_{1077} 4-bit counter
- 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?

Nope!
There exist many simple 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, Super Roboant can!

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

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

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Crumb?</th>
<th>Crumb</th>
<th>Move</th>
<th>State</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>(</td>
<td>-</td>
<td>Y</td>
<td>R</td>
<td>S1</td>
<td>Mark open paren</td>
</tr>
<tr>
<td>S1</td>
<td>)</td>
<td>-</td>
<td>Y</td>
<td>L</td>
<td>S2</td>
<td>Mark close paren</td>
</tr>
<tr>
<td>S1</td>
<td>sp</td>
<td>-</td>
<td>N</td>
<td>L</td>
<td>S4</td>
<td>Reached end</td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>L</td>
<td>S2</td>
<td>Scan back to last open</td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>R</td>
<td>S3</td>
<td>Eat crumb</td>
</tr>
<tr>
<td>S3</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>R</td>
<td>S3</td>
<td>Goto close paren</td>
</tr>
<tr>
<td>S3</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>R</td>
<td>S1</td>
<td>Eat crumb</td>
</tr>
<tr>
<td>S4</td>
<td>( or</td>
<td>N</td>
<td>N</td>
<td>L</td>
<td>S4</td>
<td>Move Left</td>
</tr>
<tr>
<td>S4</td>
<td>sp</td>
<td>-</td>
<td>Y</td>
<td>L</td>
<td>S5</td>
<td>Unmatched/Eat</td>
</tr>
<tr>
<td>S5</td>
<td>( or</td>
<td>N</td>
<td>N</td>
<td>L</td>
<td>S5</td>
<td>Unmatched/Eat</td>
</tr>
<tr>
<td>S5</td>
<td>sp</td>
<td>-</td>
<td>N</td>
<td>L</td>
<td>Halt</td>
<td>Unmatched</td>
</tr>
</tbody>
</table>

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?
Yes, Roboant can!

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Crumb?</th>
<th>Crumb Move</th>
<th>Next State</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>(</td>
<td>Y</td>
<td>R</td>
<td>S1</td>
<td>Mark open paren</td>
</tr>
<tr>
<td>S1</td>
<td>)</td>
<td>Y</td>
<td>L</td>
<td>S2</td>
<td>Mark close paren</td>
</tr>
<tr>
<td>S1</td>
<td>sp</td>
<td>-</td>
<td>N</td>
<td>L</td>
<td>S4</td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td>N</td>
<td>L</td>
<td>S2</td>
<td>Scan back to last open</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td>Y</td>
<td>N</td>
<td>R</td>
<td>S3</td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td>-</td>
<td>N</td>
<td>R</td>
<td>S3</td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td>Y</td>
<td>N</td>
<td>R</td>
<td>S1</td>
</tr>
<tr>
<td>S4</td>
<td>( or )</td>
<td>N</td>
<td>L</td>
<td>S4</td>
<td>Move Left</td>
</tr>
<tr>
<td>S4</td>
<td>( or )</td>
<td>Y</td>
<td>N</td>
<td>L</td>
<td>S5</td>
</tr>
<tr>
<td>S4</td>
<td>sp</td>
<td>-</td>
<td>Y</td>
<td>L</td>
<td>S5</td>
</tr>
<tr>
<td>S5</td>
<td>( or )</td>
<td>-</td>
<td>N</td>
<td>L</td>
<td>S5</td>
</tr>
<tr>
<td>S5</td>
<td>sp</td>
<td>-</td>
<td>N</td>
<td>L</td>
<td>Halt</td>
</tr>
</tbody>
</table>

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

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...

This simple addition solves "FINITE" problem of FSMs. (Like crumbs)
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) incremenetr.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Tape Input</th>
<th>Write Tape</th>
<th>Move</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>S0</td>
</tr>
<tr>
<td>S0</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>Halt</td>
</tr>
</tbody>
</table>

Turing Machine Tapes as Integers

Canonical names for bounded tape configurations:

\[
\begin{array}{cccccccc}
| b_8 | b_7 | b_6 | b_5 | b_4 | b_3 | b_2 | b_1 |
\end{array} \Rightarrow \begin{array}{cccccccc}
| 0   | 0   | 1   | 0   | 0   | 1   | 0   | 0   |
\end{array}
\]

Look, it’s just FSM i operating on tape j
TMs as Integer Functions

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

$$y = T_i [x]$$

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

I wonder if a TM can compute EVERY integer function...

Alternative Models of Computation

Turing Machines [Turing]

Recursive Functions [Kleene]

F(0,x) = x
F(1+y,x) = 1+F(x,y)

(\text{define } (\text{fact } n) \\
\text{(\ldots (fact } (- n 1)) \ldots))

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

\lambda.x.\lambda.y.xyz

(lambda (x) (lambda (y) (lambda (z) (x (y) (z)))))
The 1st Computer Industry Shakeout

Here's a TM that computes SQUARE ROOT!

FSM

And the Battles Raged

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

\((\lambda(x) \ldots)\)

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

\((\lambda(x \ n) \ldots)\)
**Fundamental Result:**

**Computable Functions**

Each model is capable of computing exactly the same set of integer functions!

Proof Technique: Constructions that translate between models

**BIG IDEA:**

Computability, independent of computation scheme chosen

**Church’s Thesis:**

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

---

**Computable Functions**

\[ f(x) \text{ computable} \iff \text{for some } k, \text{ all } x: \]

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

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

\(<x,y> = \text{integer whose even bits come from } x, \text{ and whose odd bits come from } y; \)

whence

\[ f_k(x, y) \equiv T_k[<x, y>] \]

\[ f_{123456}(x,y) = x \times y \]

\[ f_{23456}(x) = 1 \text{ iff } x \text{ is prime, else } 0 \]
### 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.

<table>
<thead>
<tr>
<th>f_i</th>
<th>f_i(0)</th>
<th>f_i(1)</th>
<th>f_i(2)</th>
<th>...</th>
<th>f_i(j)</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_0</td>
<td>37</td>
<td>23</td>
<td>*</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>f_1</td>
<td>62</td>
<td>*</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>f_k</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>f_k(j)</td>
<td></td>
</tr>
</tbody>
</table>

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

### The Halting Problem

The Halting Function:

\[ T_H[k, j] = 1 \text{ iff } TM[k][j] \text{ halts, else } 0 \]

Can a Turing machine compute this function?

Suppose, for a moment, \( T_H \) exists:

1 iff \( T_x[y] \) HALTS
0 otherwise

Then we can build a \( T_{Nasty} \):

Replace the Halt state of \( T_H \) with this.

If \( T_{Nasty} \) is computable, then so is \( T_{Nasty} \).
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!

Net Result: 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.

Reality: Limits of Turing Machines

A Turing machine is a formal abstraction that addresses
- Fundamental Limits of Computability – What it means to compute. The existence of incomputable functions.
- We know of no machine that is more powerful than a Turing machine in terms of the functions 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 120)