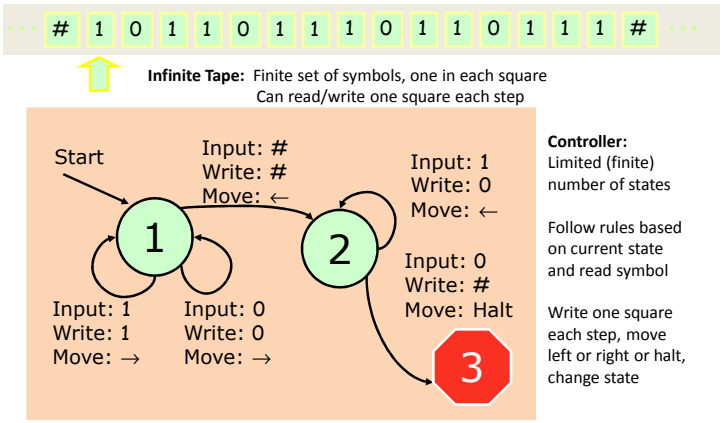
Class 39: Universality

cs1120 Fall 2009
David Evans
University of Virginia

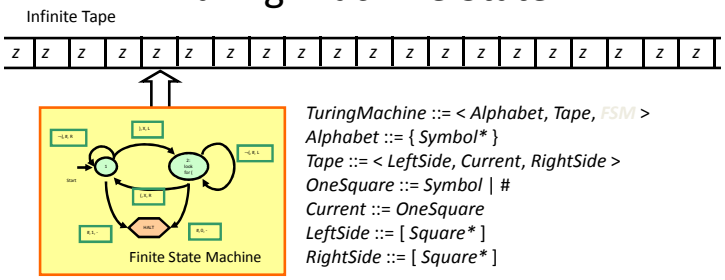
Menu

- Recap Turing Machine from last class
- Halting Problem for Turing Machines:
  - Proof that the Halting Problem for TMs is noncomputable
  - Universal Turing Machines: one TM that can simulate every other TM
- Church-Turing Thesis: all reasonable computing models are equivalent!
- Lambda Calculus: an alternate computing model, equivalent to Turing Machine

Recap: Turing's Model



Turing Machine State



TuringMachine ::= < *Alphabet*, *Tape*, *FSM* >
Alphabet ::= { *Symbol** }
Tape ::= < *LeftSide*, *Current*, *RightSide* >
OneSquare ::= *Symbol* | #
Current ::= *OneSquare*
LeftSide ::= [*Square**]
RightSide ::= [*Square**]

Everything to left of *LeftSide* is **#**.
Everything to right of *RightSide* is **#**.

Although we typically draw TMs as pictures, we could write them down as strings in this language (which could be converted to whole numbers)

Enumerating Turing Machines

- Now that we've decided how to describe Turing Machines, we can number them
- TM-5023582376 = balancing parens
- TM-57239683 = even number of 1s
- TM-3523796834721038296738259873 = Universal TM
- TM-3672349872381692309875823987609823712347823 = WindowsXP

Not the real numbers – they would be much much much much much bigger!

Halting Problem

Pythonic Halting Problem

Input: a string describing a Python program, *P*

Output: if evaluating *P* would eventually finish, output **True**; otherwise, output **False**.

TM Halting Problem

Input: a string describing a Turing Machine, *M*

Output: if *M* would eventually Halt, output **True**; otherwise, output **False**.

Halting Problem "Proof"

Pythonic Halting Problem

```
def paradox():
  if halts('paradox()'):
    while True:
      pass
```

TM Halting Problem

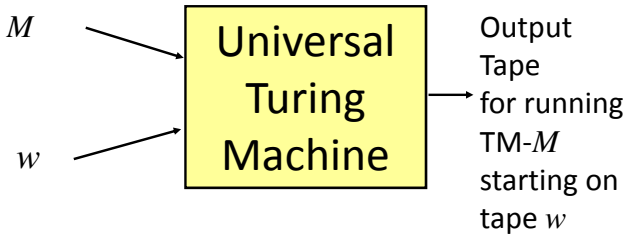
HALTS(*M*) = TM that solves TM Halting Problem: input tape describes TM *M*, output tape: #1 if *M* halts, otherwise #0

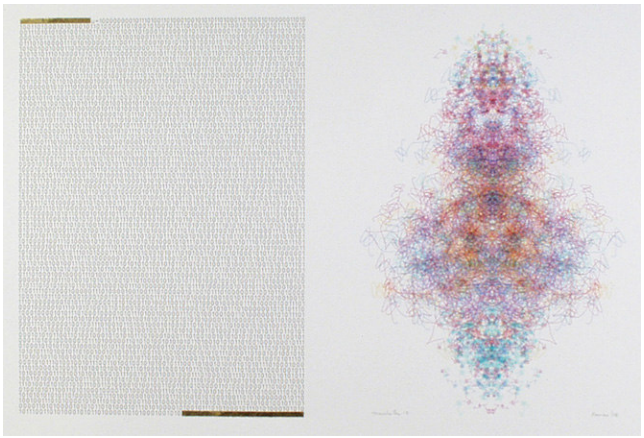
PARADOX = TM that:
1. simulates **HALTS(PARADOX)**
2. Then, if tape is #1, loop forever; if tape is #0, **HALT**

simulates? This proof assumes we can design a TM that simulates any other TM!

An Any TM Simulator

Input: < Description of some TM *M*, *w* >
Output: result of running *M* on *w*

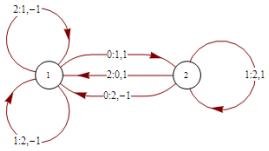




Manchester Illuminated Universal Turing Machine, #9
from http://www.verostko.com/manchester/manchester.html

Universal Computing Machine

2-state, 3-symbol Turing machine proved universal by Alex Smith in 2007



SCIENTIFIC AMERICAN

News - October 25, 2007

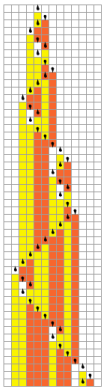
***A New Kind of Science* Author Pays Brainy Undergrad $25,000 for Identifying Simplest Computer**
But will it jumpstart Stephen Wolfram's scientific revolution?

By JR Minkel

Five years ago, grown-up wunderkind Stephen Wolfram did his darnedest to alter the course of scientific history. The former particle physicist, who is by all accounts a genius, poured two decades worth of heady thoughts on the nature of computers, mathematics and the universe into an ambitious 1,200-page tome modestly entitled *A New Kind of Science*.

And when that didn't sway his peers, he offered cash.

Alex Smith,
University of Birmingham



What This Means

- If you can:
  - Keep track of a finite state
  - Follow transition rules
  - Read and write to memory

  you can simulate a universal Turing machine.
- A Turing machine can simulate the world's most powerful supercomputer
  - Thus, your cell phone can simulate the world's most powerful supercomputer (it'll just take a lot longer and will run out of memory)
- No computer that can be simulated by a TM can solve the Halting Problem

Church-Turing Thesis

- All mechanical computers are equally powerful (except for practical limits like memory size, time, display, energy, etc.)
- There exists some Turing machine that can simulate *any* mechanical computer
- *Any* computer that is powerful enough to simulate a Turing machine, can simulate any mechanical computer

Alonzo Church, 1903-1995

Alan Turing, 1912-1954

Church's Computing Model: Lambda Calculus

- Developed in 1930s in attempt to formalize mathematics (similar to Bertrand Russell's goals)
- Became model of computing
- Basis of LISP and Scheme

What is Calculus?

$d/dx\ x^n = nx^{n-1}$ [Power Rule]

$d/dx\ (f + g) = d/dx\ f + d/dx\ g$ [Sum Rule]

Calculus is a branch of mathematics that deals with limits and the differentiation and integration of functions of one or more variables...

Real Definition

- A *calculus* is just a bunch of rules for manipulating symbols.
- People can give meaning to those symbols, but that's not part of the calculus.
- *Differential calculus* is a bunch of rules for manipulating symbols. There is an interpretation of those symbols corresponds with physics, slopes, etc.

Lambda Calculus

- Rules for manipulating strings of symbols

  $term =\ variable$
  $\mid term\ term$
  $\mid (term)$
  $\mid \lambda\ variable\ .\ term$

- One main rule: β-reduction

  $((\lambda x.\ M)N) \Rightarrow_\beta M$

  with all x's in M replaced by N

Note: I have slightly altered the traditional syntax to make it more like Scheme.

This is a lot like Scheme, without primitives, special forms, mutation!

Lambda Calculus

- Developed in 1930s in attempt to formalize mathematics (similar to Bertrand Russell's goals)
- Original attempt was inconsistent!
  - Kleene-Rosser Paradox:

    $k = \lambda x\ .\neg(x\ x)$

    $(k\ k) = \lambda x\ .\neg(x\ x)k = \neg(k\ k)$
- Church's solution: model **computation** not logic

  $(k\ k)$ is a nonterminating recursive definition, not a contradiction!

  $(k\ k) \Rightarrow \neg(k\ k) \Rightarrow \neg\neg(k\ k) \Rightarrow \neg\neg\neg\ (k\ k) \Rightarrow \ldots$

Evaluating Lambda Expressions

- ***redex***: Term of the form $((\lambda x.\ M)N)$

  Something that can be β-reduced
- An expression is in ***normal form*** if it contains no redexes.
- To evaluate a lambda expression, keep doing reductions until you get to *normal form*.

Some Simple Functions

$\mathbf{I} \equiv \lambda x.x$

$\mathbf{C} \equiv \lambda x.(\lambda y.\ yx)$

$((\mathbf{C}\ \mathbf{I})\ \mathbf{I}) = (((\lambda x.(\lambda y.\ yx))\ (\lambda x.x))\ (\lambda x.x))$

$\rightarrow_\beta\ ((\lambda y.\ y\ (\lambda x.x))\ (\lambda x.x))$

$\rightarrow_\beta\ (\lambda x.x\ (\lambda x.x))$

$\rightarrow_\beta \lambda x.x$

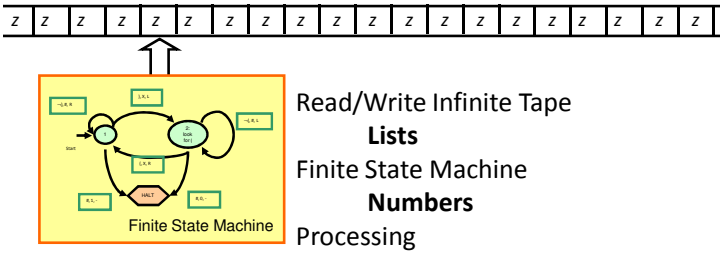
$\equiv \mathbf{I}$

Power of Lambda Calculus

- Is Lambda Calculus a *universal language*?
  - Can we compute any computable algorithm using Lambda Calculus?
- Option 1: Prove it is **not**
  - Find some Turing Machine that cannot be simulated with Lambda Calculus
- Option 2: Prove it **is:**
  - Show you can simulate *every* Turing Machine using Lambda Calculus

Since a Universal Turing Machine can simulate every other TM, it is enough to show we can simulate a Universal Turing Machine using Lambda Calculus

Simulating a Turing Machine



Read/Write Infinite Tape

Lists

Finite State Machine

Numbers

Processing

Way to make decisions (if)

Way to keep going

Continued on Friday...