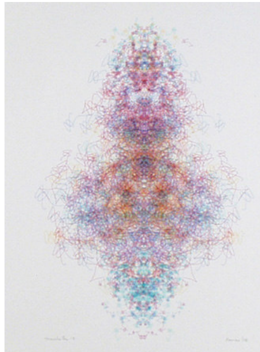
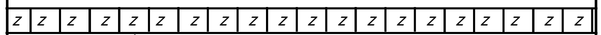
Class 27: Universal Turing Machines

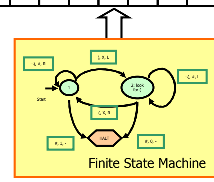


CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

Turing Machine (1936)





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

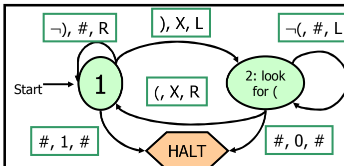
Everything to left of *LeftSide* is **#**.
Everything to right of *RightSide* is **#**.

Describing Finite State Machines

TuringMachine ::= < *Alphabet*, *Tape*, ***FSM*** >
FSM ::= < *States*, *TransitionRules*, *InitialState*, *HaltingStates* >
States ::= { *StateName** }
InitialState ::= *StateName* must be element of *States*
HaltingStates ::= { *StateName** } all must be elements of *States*
TransitionRules ::= { *TransitionRule** }
TransitionRule ::=
< *StateName*, ;; Current State
OneSquare, ;; Current square
StateName, ;; Next State
OneSquare, ;; Write on tape
Direction > ;; Move tape
Direction ::= L, R, #

Transition Rule is a procedure:
Inputs: *StateName*, *OneSquare*
Outputs: *StateName*, *OneSquare*, *Direction*

Example Turing Machine



TuringMachine ::= < *Alphabet*, *Tape*, ***FSM*** >
FSM ::= < *States*, *TransitionRules*, *InitialState*, *HaltingStates* >
Alphabet ::= { (,), X }
States ::= { 1, 2, HALT }
InitialState ::= 1
HaltingStates ::= { HALT }
TransitionRules ::= { < 1,), 2, X, L >,
< 1, #, HALT, 1, # >,
< 1, ¬), #, R >,
< 2, (, 1, X, R >,
< 2, #, HALT, 0, # >,
< 2, ¬), #, L >,}

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 = Photomosaic Program
- TM-3672349872381692309875823987609823712347823 = WindowsXP

Not the real numbers – they would be much bigger!

Universal Turing Machine

P
Number of TM

I
Input Tape

also, just a number!

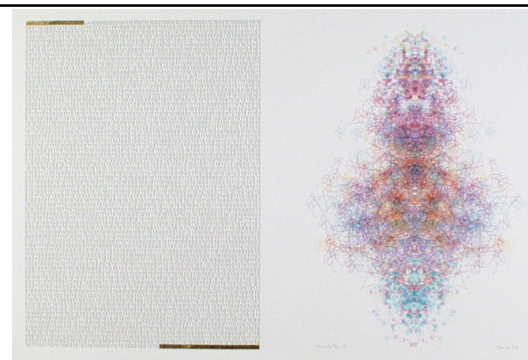
→ **Universal Turing Machine** →

Output Tape for running TM-*P* in tape *I*

Can we make a Universal Turing Machine?

Yes!

- People have designed Universal Turing Machines with
  - 4 symbols, 7 states (Marvin Minsky)
  - 4 symbols, 5 states
  - 2 symbols, 22 states
  - 18 symbols, 2 states
  - 2 states, 5 symbols (Stephen Wolfram)
- No one knows what the smallest possible UTM is



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

Church-Turing Thesis

- Any mechanical computation can be performed by a Turing Machine
- There is a TM-*n* corresponding to every decidable problem
- We can simulate one step on any "normal" (classical mechanics) computer with a constant number of steps on a TM:
  - If a problem is in P on a TM, it is in P on an iMac, CM5, Cray, Palm, etc.
  - But maybe not a quantum computer! (later class)

Universal Language

- Is Scheme as powerful as a Universal Turing Machine?
- Is a Universal Turing Machine as powerful as Scheme?

Complexity in Scheme

- Special Forms
  - **if**, **cond**, **define**, etc.
- Primitives
  - Numbers (infinitely many)
  - Booleans: **#t**, **#f**
  - Functions (**+**, **-**, **and**, **or**, etc.)
- Evaluation Complexity
  - Environments (more than ½ of our eval code)

If we have lazy evaluation and don't care about abstraction, we don't need these.

Hard to get rid of?

Can we get rid of all this and still have a useful language?

λ-calculus

Alonzo Church, 1940

(LISP was developed from λ-calculus, not the other way round.)

term = *variable*
| *term term*
| (*term*)
| **λ** *variable* **.** *term*

What is Calculus?

- In High School:

$d/dx\ x^n = nx^{n-1}$ [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 in the language:

  term = *variable*
  | *term term*
  | (*term*)
  | λ *variable* . *term*

- Humans can give meaning to those symbols in a way that corresponds to computations.

Why?

- Once we have precise and formal rules for manipulating symbols, we can use it to reason with.
- Since we can interpret the symbols as representing computations, we can use it to reason about programs.

Evaluation Rules

α-reduction (renaming)

$\lambda y.\ M \Rightarrow_\alpha \lambda v.\ (M\ [y\ \alpha\ \ v])$

where v does not occur in M.

β-reduction (substitution)

$(\lambda x.\ M)N \Rightarrow_\beta M\ [\ x\ \alpha\ \ N\ ]$

Reduction (Uninteresting Rules)

$\lambda y.\ M \rightarrow \lambda v.\ (M\ [y\ \alpha\ \ v])$

where v does not occur in M.

$M \rightarrow M$

$M \rightarrow N \Rightarrow PM \rightarrow PN$

$M \rightarrow N \Rightarrow MP \rightarrow NP$

$M \rightarrow N \Rightarrow \lambda x.\ M \rightarrow \lambda x.\ N$

$M \rightarrow N$ and $N \rightarrow P \Rightarrow M \rightarrow P$

β-Reduction
(the source of all computation)

$$(\lambda x.\, M)N \rightarrow M\,[\, x\,\alpha \;\; N\,]$$

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 (*redices*).
- To evaluate a lambda expression, keep doing reductions until you get to normal form.

Recall Apply in Scheme

"To **apply** a procedure to a list of arguments, **eval**uate the procedure in a new environment that binds the formal parameters of the procedure to the arguments it is applied to."

- We've replaced environments with substitution.
- We've replaced **eval** with reduction.

Some Simple Functions

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

$\mathbf{C} \equiv \lambda xy.yx$

Abbreviation for $\lambda x.(\lambda y.\, yx)$

$\mathbf{CII} = (\lambda x.(\lambda y.\, yx))\ (\lambda x.x)\ (\lambda x.x)$

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

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

$\rightarrow_\beta \lambda x.x$

$= \mathbf{I}$

Example

$$\lambda\, f.\ ((\lambda\ x.f\,(xx))\ (\lambda\ x.\ f\,(xx)))$$

Try this one at home...

Institute of Mathematical Science presents:

Professor, Author, Newspaper Columnist, and Public Speaker

John Allen Paulos

A Mathematician Reads the Newspaper

His irreverent and pointed comments entertain as well as educate...
--Charles Seife, *Philadelphia Inquirer*

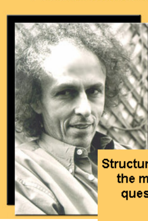
...full of fun, full of information, full of insights.
--Peter Hilton, *American Math. Monthly*

...wise little book...
--Joel Achenbach, *Washington Post*

...provides a wide-ranging collection of musings on mathematics, the media and life itself...
--Jon Van, *Chicago Tribune*

...that mysterious masked mathematician on a white horse...
--Molly Ivins, Syndicated Columnist

...irresistible...
--Rudy Rucker, *Scientific American*



Structured like a newspaper, the talk and the book on which it's based investigate the mathematical angles of stories in the news and offer novel perspectives, questions, and ideas to those who can't get along without their daily paper.

Wednesday, October 26, 7:30pm
Physics Building, Room 203
University of Virginia, Charlottesville, Virginia

Charge

- PS6 Due Monday
- PS7/PS8 Out Monday
  - PS8: "Make a dynamic web application"
  - PS7: Learn to use tools you will use for PS8
  - If you have a group and idea in mind for PS8 soon enough, you may not need to do PS7
- Friday:
  - Computability in Theory and Practice
  - Making Primitives using Lambda Calculus