Lecture 37: A Universal Computer

PS8 returned at your project design review meetings

Remember to email your project descriptions before midnight tonight

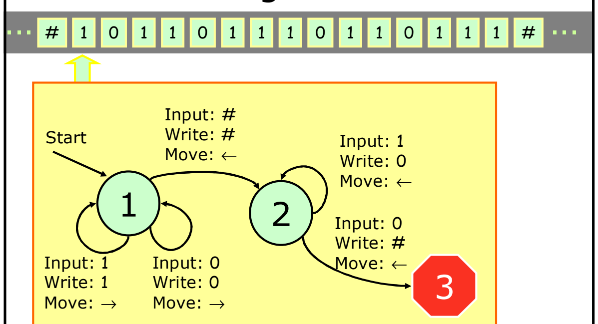
CS150: Computer Science
University of Virginia
Computer Science

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

Turing Machine: FSM + Infinite Tape

- Start:
  - FSM in Start State
  - Input on Infinite Tape
  - Tape head at start of input
- Step:
  - Read current input symbol from tape
  - Follow transition rule from current state on input
    - Write symbol on tape
    - Move L or R one square
    - Update FSM state
- Finish: Transition to halt state

Turing Machine



Adding

- Input on tape:

  ...#$n_k n_{k-1} ... n_0 + m_l m_{l-1} ... m_0$#.....

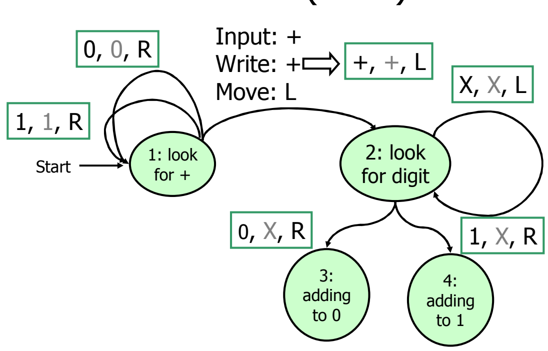
  – Number represented in binary
- Output:

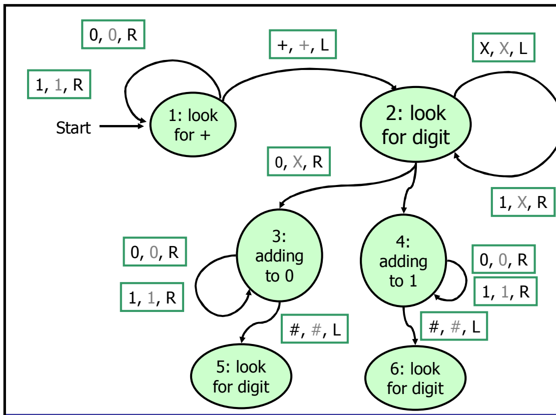
  ...#$r_d r_{d-1} ... r_0$#.....

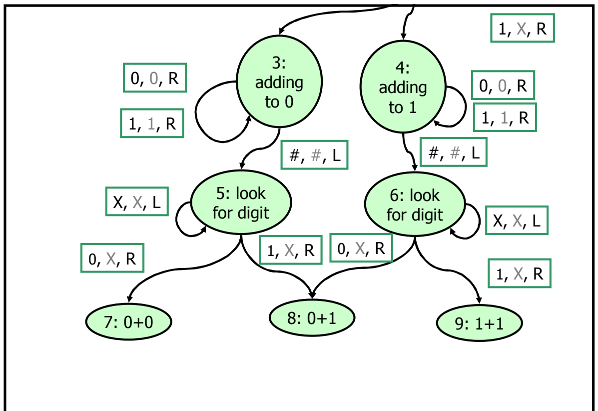
  where $r = n + m$

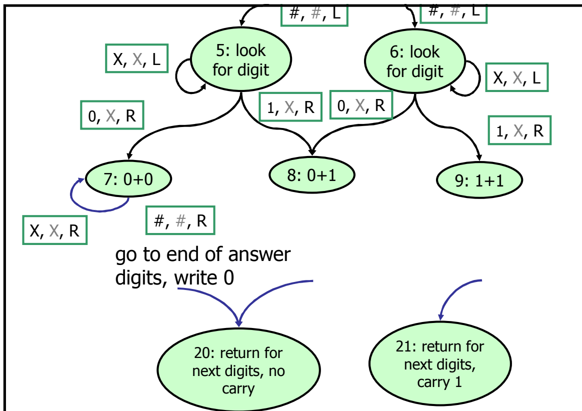
Can we implement addition with a TM?

Adder TM (Start)

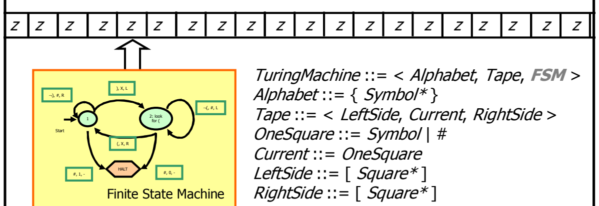








Turing Machine



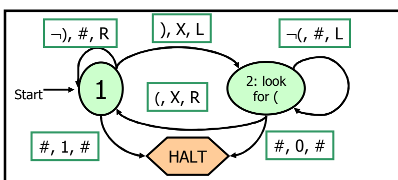
TuringMachine ::= < *Alphabet*, *Tape*, ***FSM*** >
Alphabet ::= { *Symbol** }
Tape ::= < *LeftSide*, *Current*, *RightSide* >
OneSquare ::= *Symbol* | #
Current ::= *OneSquare*
LeftSide ::= [*Square**]
RightSide ::= [*Square**]

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!

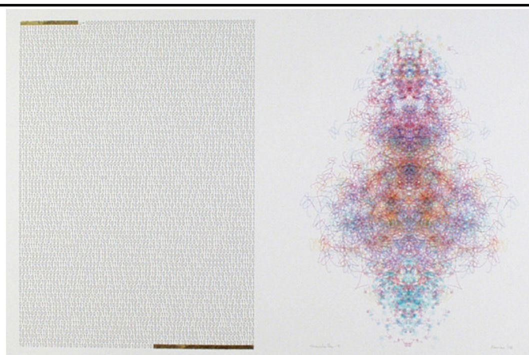
Universal Turing Machine

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 computable problem
- We can any "normal" (classical mechanics) computer with a TM
  - If a problem is in polynomial time on a TM, it is in polynomial time on an iMac, Cray, Palm, etc.
  - But maybe not a quantum computer! (later class)

Universal Language

- Is Scheme/Charme/Python as powerful as a Universal Turing Machine?

Yes: show we can simulate a UTM with a Scheme program

- Is a Universal Turing Machine as powerful as Scheme/Charme/Python?

Can we simulate a Scheme interpreter with a TM?

Complexity in Scheme

- Special Forms
  - **if**, **cond**, **define**, etc.

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

- Primitives
  - Numbers (infinitely many)
  - Booleans: **#t**, **#f**
  - Functions (**+**, **-**, **and**, **or**, etc.)

Hard to get rid of?

- Evaluation Complexity
  - Environments (more than ½ of our eval code)

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)

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

Charge

- Project Descriptions due before midnight tonight
- Exam 2 due Friday at 12:02 pm (beginning of class)
- Friday's class: student talks about research and industry