Turing machines

the simplest machines that can get the job done

Part 3 outline

- Turing machines as simple but powerful models of computation
- Extensions to the basic TM model
- JFLAP and Kara simulations
- Universal Turing machines
- Context-sensitive and unrestricted grammars
- Other equally powerful formal models of computation
- The diagonalization method and related paradoxes
- The halting problem and other undecidable problems
- Introduction to computational complexity
- Non-conventional models of computation

Turing machine

- A finite-state automaton that uses its tape as a form of storage, i.e., the tape is re-writable
- The tape is infinitely long, so a TM effectively has infinite memory
- The read/write head is also bi-directional – after each step, the head can move left, right or stay in place
- Despite its simplicity, it can do anything a real computer can do – hence, it captures the very essence of algorithms and computability

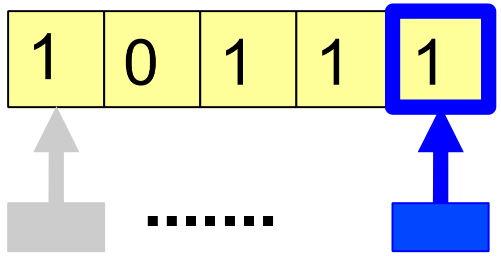
Some types of Turing machines

- **Decider TM** – for an input string x and a language L, determine if x is a member in L
  - have 2 final states: $q_{\textbf{accept}}$ and $q_{\textbf{reject}}$

- **Transducer TM** – for an input string x and a string function f, where f: $\Sigma^* \rightarrow \Sigma^*$, compute f(x), and leave the resulting string f(x) on the tape

Example: the successor function

- Given the binary alphabet $\Sigma = \{0,1\}$ and any non-empty input string x over Σ representing a binary number, compute f(x) = x+1
- Algorithm:

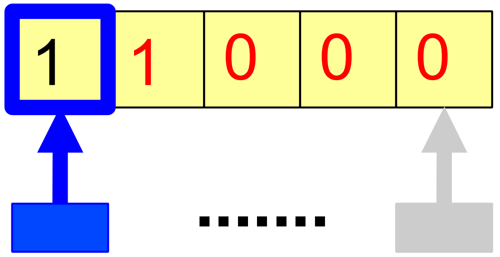


move to the right-most bit

flip 1's to 0's and move to next bit on the left, repeat this step until we reach a 0 or a blank

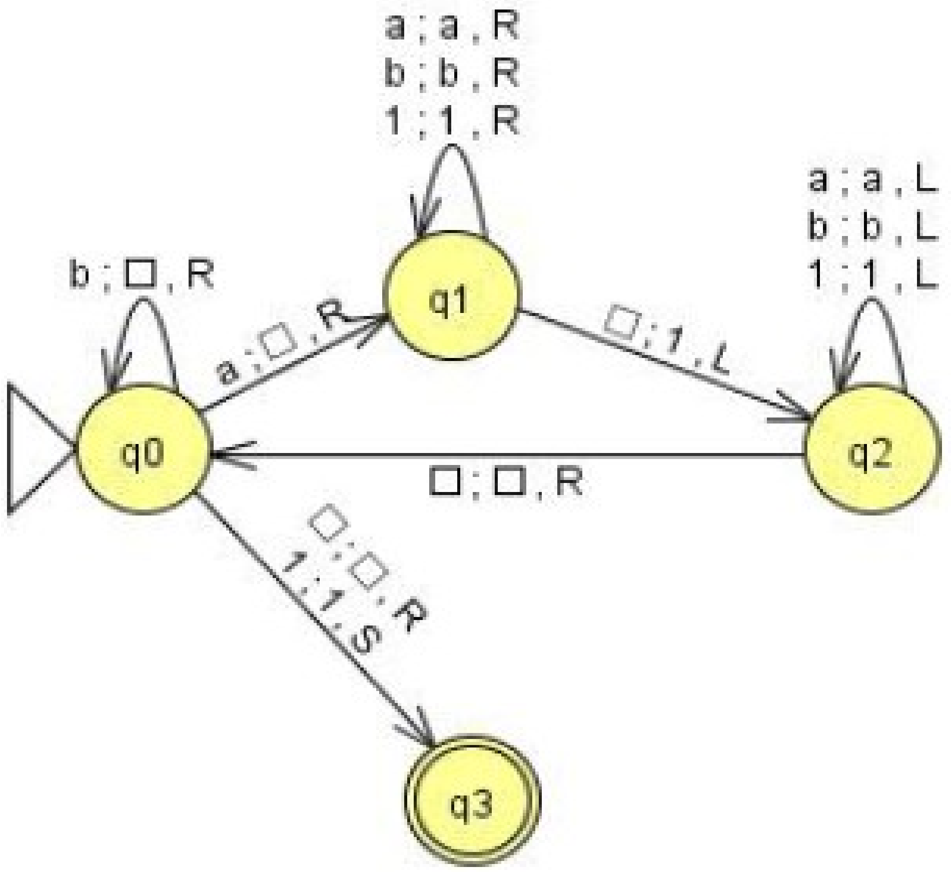
replace the 0 or the blank with a 1

move to the left-most bit



Another example, counting a's

- $f(w)$ = number of a's in w, $\Sigma = \{a,b\}$
- Examples:
  - f(aabab) = 111
  - f(bbbaab) = 11

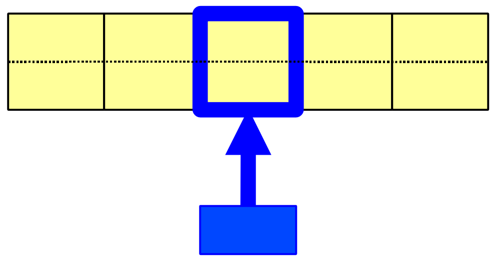


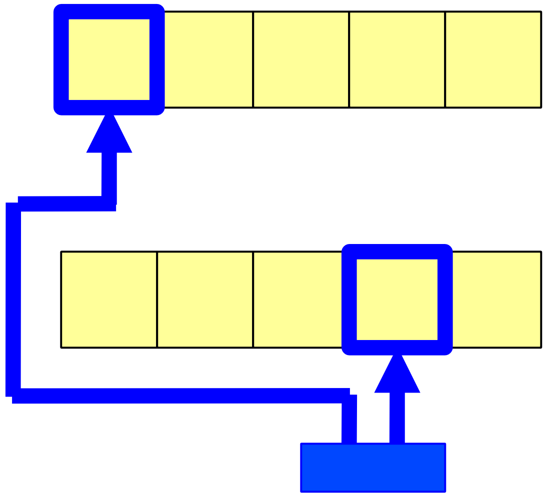
From Susan H. Rodger, An interactive approach to automata & formal languages with JFLAP

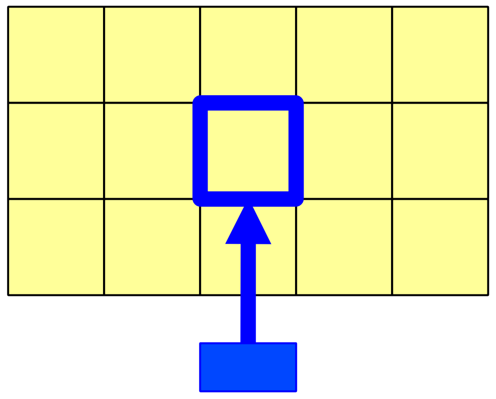
“Extensions” to the basic TM model

- **Allow multiple tracks**
  - like having n-bits in each byte
- **Allow multiple tapes with multiple independent heads**
  - like having parallel processors
- **Allow 2-dimensional “tapes”**
  - Kara is a nice simulation tool that uses a two-dimensional Turing machine model
- **Allow non-determinism**

None of these “extensions” add real power – they merely simplify the programming process







Changes to the transition function

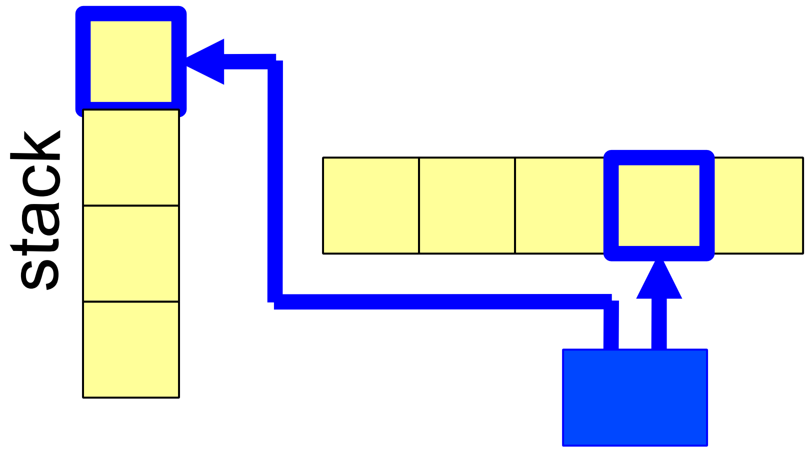
Basic model	$\delta : Q \times \Sigma \to Q \times \Sigma \times \{ L, S, R \}$
k-tracks	$\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{ L,S,R \}$
p-tapes	$\delta : Q \times \Sigma^p \to Q \times (\Sigma \times \{ L,S,R \})^p$
2-dimensional	$\delta : Q \times \Sigma \to Q \times \Sigma \times \{ L,S,R,U,D \}$
Non-deterministic	$\delta : Q \times \Sigma \to 2^{(Q \times \Sigma \times \{ L,S,R \})}$

Up

Left **Stay** **Right**

Down

TMs extend PDAs



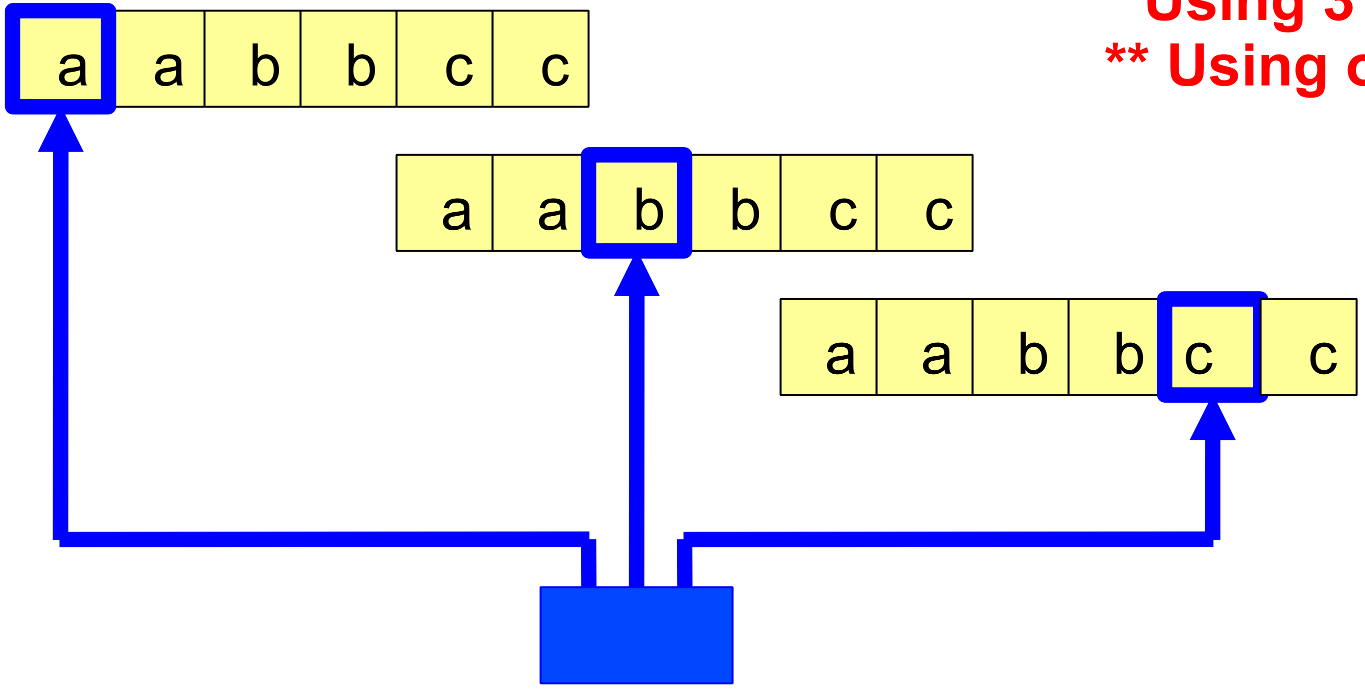
- Think of a pushdown automaton implemented as a two-tape Turing machine with one of the tapes acting as the stack

pushdown ...

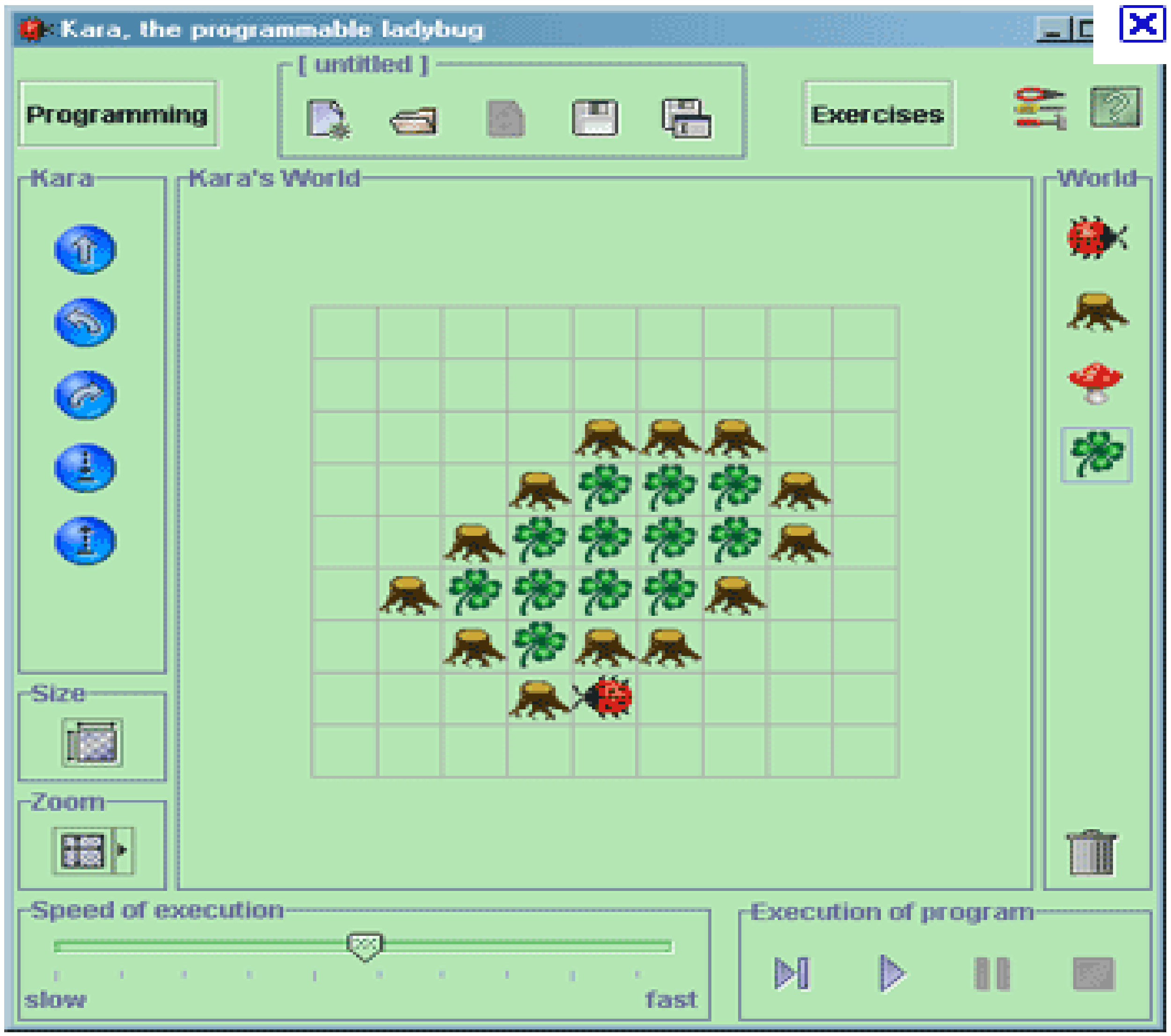
Some non-CFLs can be recognized by TMs

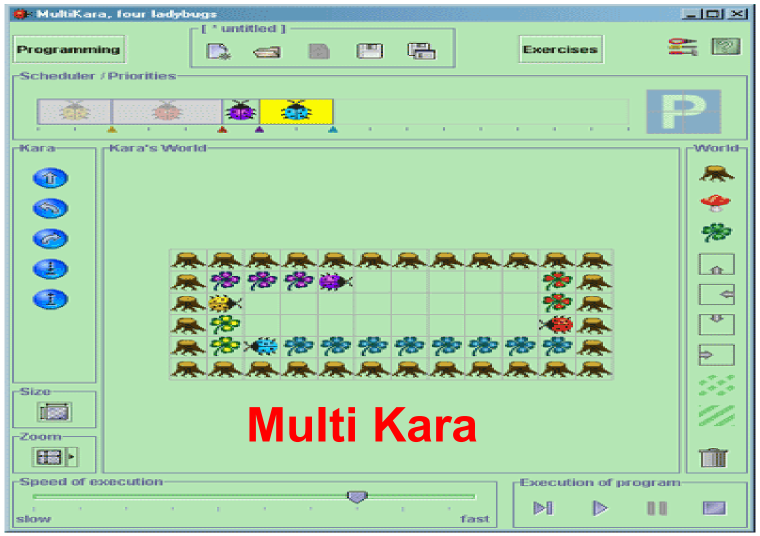
- Any language that can be recognized by a PDA can also be recognized by a TM
- But TMs are more powerful than PDAs
- How can a TM recognize the non-CFL L = $\{a^n b^n c^n : n>0\}$?

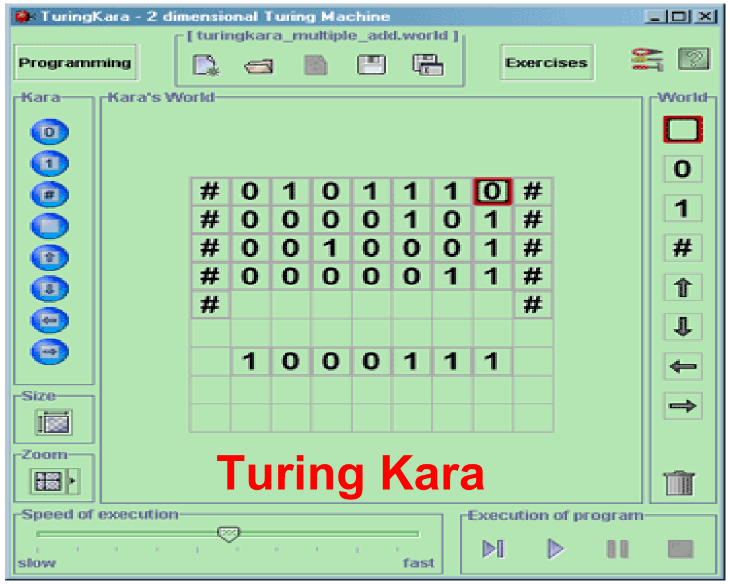
* Using 3 tapes?
** Using only 1 tape?



2D Turing machines with Kara







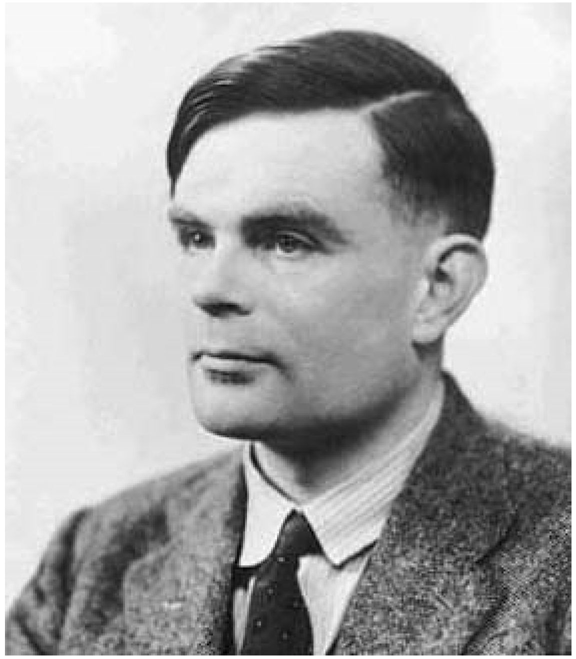
```
# 0 1 0 1 1 1 0 #
# 0 0 0 0 1 0 1 #
# 0 0 1 0 0 0 1 #
# 0 0 0 0 0 1 1 #
#               #

  1 0 0 0 1 1 1
```

http://www.swisseduc.ch/compscience/karatojava/

Universal Turing machines

- Our example Turing machines seem to solve very specific problems
- It is possible however, to define a TM that can simulate any TM on any input – this makes Turing machines general-purpose problem solvers


Alan Turing
1912-1954

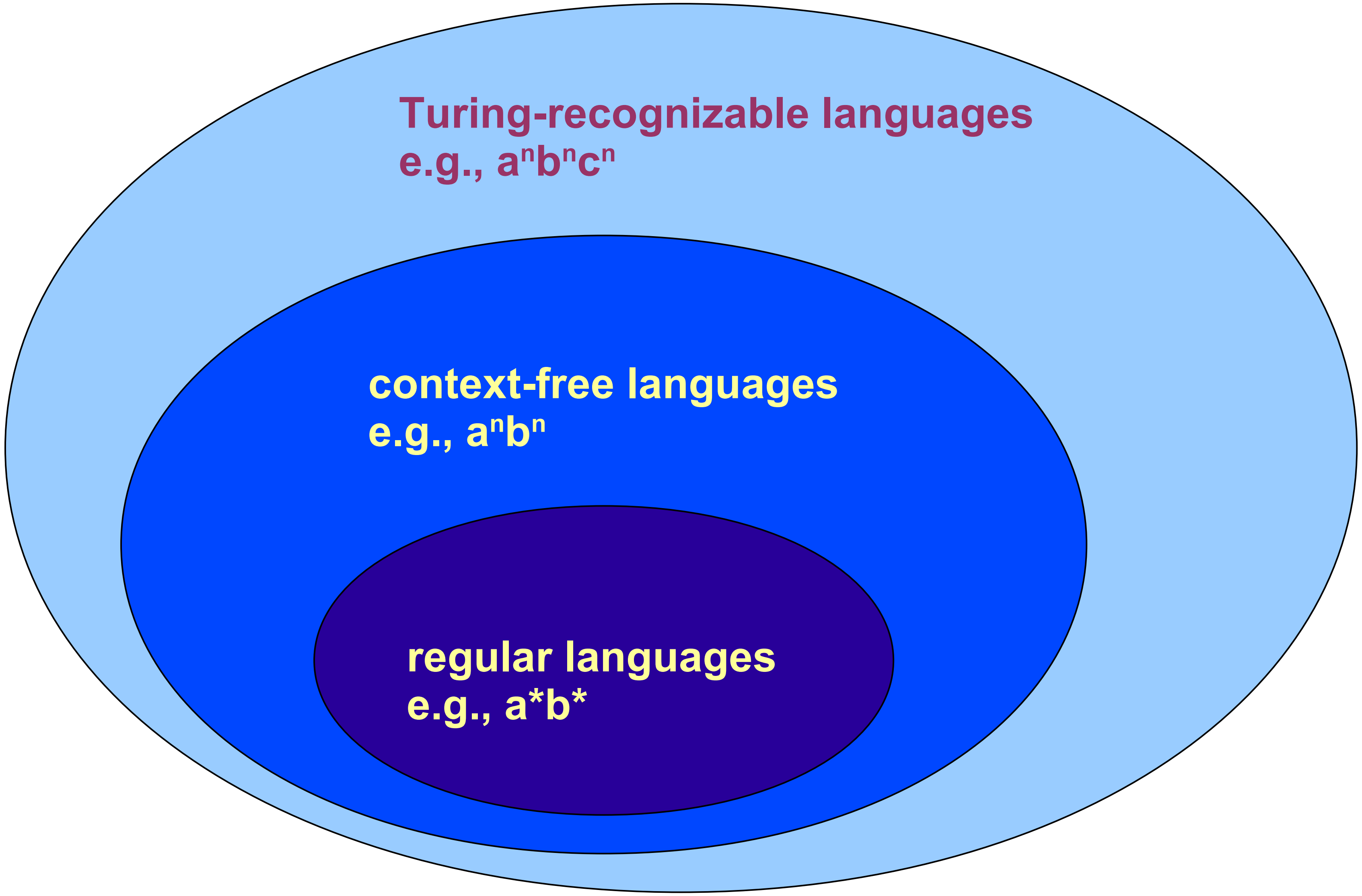
Universal Turing Machines

- UTMs can be considered as the origin of the stored-program computer
- UTMs are <u>programmable</u> to simulate any other TM; user programs serve as data to the compiler or interpreter
- Despite its simplicity, Kara's UTM is actually quite big – there is a UTM with only 7 states and 4 symbols (Minsky 1962) and another with only 2 states and 5 symbols (Wolfram 1985)

Regular $\subset$ CFL $\subset$ Turing-recognizable

DFA $\subset$ PDA $\subset$ Turing Machine

Regular Grammar $\subset$ CFG $\subset$ Unrestricted Grammar



Unrestricted Grammars

- Every rule in the grammar has the form $\alpha \rightarrow \beta$, where α and β can be any sequence of terminals or variables
- Example rule: bXc $\rightarrow$ aBcDeF
- When $|\alpha| \leq |\beta|$ in all the rules, we have a context-sensitive grammar (corresponds to linear bounded automata, a weaker form of Turing machines)
- regular $\subset$ context-free $\subset$ context-sensitive $\subset$ unrestricted
- Context-sensitive grammars and unrestricted grammars are more powerful than CFGs, they can generate many non-context-free languages

A grammar for equal # of a's, b's and c's

- We want a grammar for the language
  L = {abc, acb, bac, ..., aabbcc, abcabc, … }
- Exercise: Show that this language cannot be regular nor context-free

$S \rightarrow ABCS$ (generate $(ABC)^+$ first)

$AB \rightarrow BA$ (swap)	$A \rightarrow a$ (convert to terminals)
$AC \rightarrow CA$	$B \rightarrow b$
$BC \rightarrow CB$	$C \rightarrow c$

- Exercise: Can you revise this to generate the language
  $L_2 = \{ a^n b^n c^n : n > 0 \}$? Note that $L_2 \subset L$.

Grammar for L = { a^n: n is a positive power of 2 }

- We want an unrestricted grammar that would generate the language L = { a^2, a^4, a^8, a^{16}, … }
- Exercise: Show that this language cannot be regular nor context-free
- An unrestricted grammar for L: (Hopcroft & Ullman)

$S \rightarrow \langle Ca \rangle$	(<,> are endmarkers)	$aD \rightarrow Da$
$Ca \rightarrow aaC$	(C is a doubler)	$\langle D \rightarrow \langle C$
$C\rangle \rightarrow D\rangle$	(prepare to double again)	$aE \rightarrow Ea$
$C\rangle \rightarrow E$	(stop when enough)	$\langle E \rightarrow \varepsilon$

Linear derivations for aa, aaaa

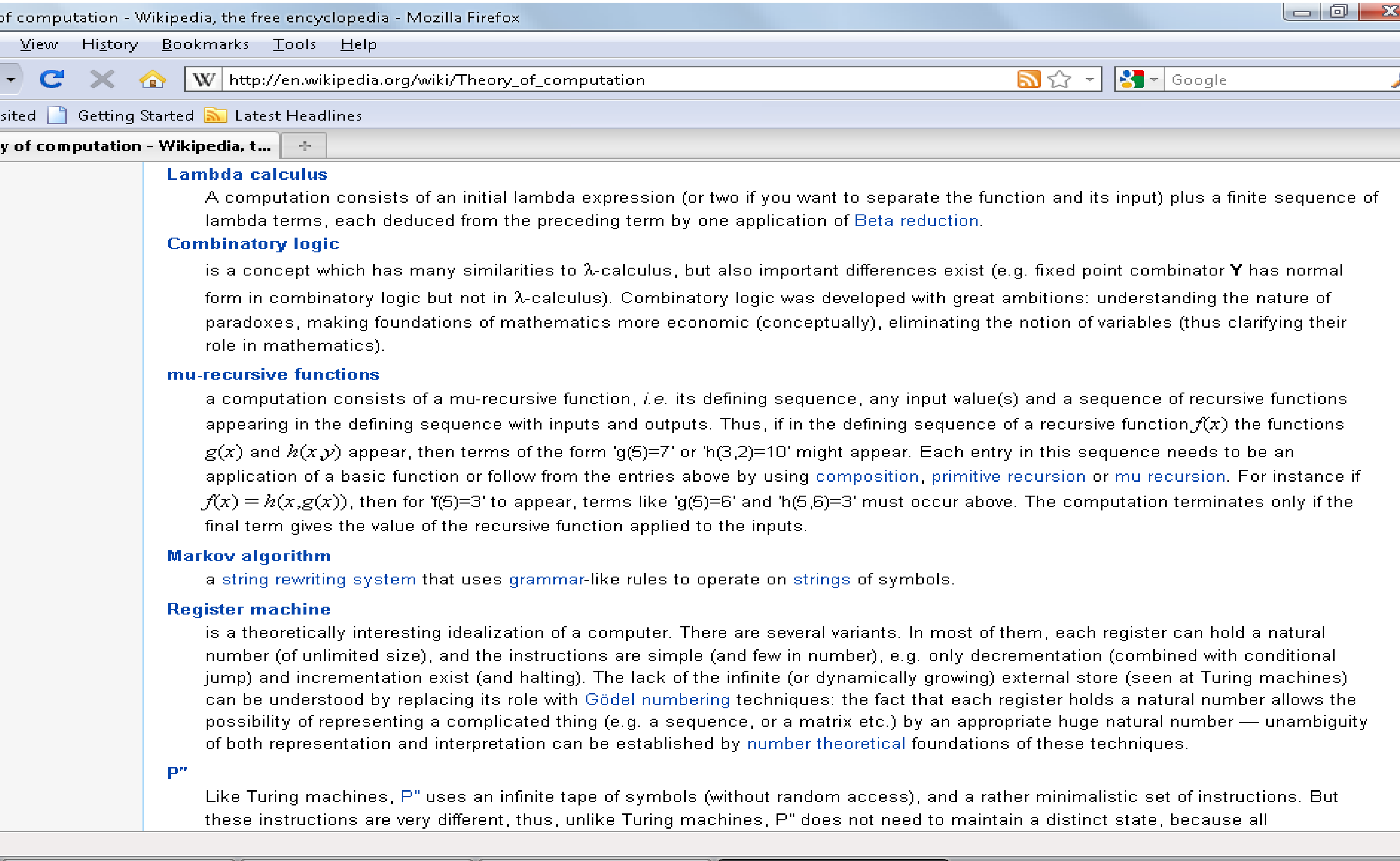
S → <Ca>	(<,> are endmarkers)	aD → Da
Ca → aaC	(C is a doubler)	<D → <C
C> → D>	(prepare to double again)	aE → Ea
C> → E	(stop when enough)	<E → ε

$S \Rightarrow_1 \text{<Ca>} \Rightarrow_2 \text{<aaC>} \Rightarrow_4 \text{<aaE} \Rightarrow_7 \text{<aEa} \Rightarrow_7 \text{<Eaa} \Rightarrow_8 \text{aa}$

$S \Rightarrow_1 \text{<Ca>} \Rightarrow_2 \text{<aaC>} \Rightarrow_3 \text{<aaD>} \Rightarrow_5 \text{<aDa>} \Rightarrow_5 \text{<Daa>}$

$\Rightarrow_6 \text{<Caa>} \Rightarrow_2 \text{<aaCa>} \Rightarrow_2 \text{<aaaaC>} \Rightarrow^* \text{<aaaaa>} \Rightarrow_8 \text{aaaa}$

Other “equally powerful” formal models


of computation - Wikipedia, the free encyclopedia - Mozilla Firefox
View History Bookmarks Tools Help
http://en.wikipedia.org/wiki/Theory_of_computation
Google
sited Getting Started Latest Headlines
y of computation - Wikipedia, t...

Lambda calculus
A computation consists of an initial lambda expression (or two if you want to separate the function and its input) plus a finite sequence of lambda terms, each deduced from the preceding term by one application of Beta reduction.

Combinatory logic
is a concept which has many similarities to λ-calculus, but also important differences exist (e.g. fixed point combinator **Y** has normal form in combinatory logic but not in λ-calculus). Combinatory logic was developed with great ambitions: understanding the nature of paradoxes, making foundations of mathematics more economic (conceptually), eliminating the notion of variables (thus clarifying their role in mathematics).

mu-recursive functions
a computation consists of a mu-recursive function, *i.e.* its defining sequence, any input value(s) and a sequence of recursive functions appearing in the defining sequence with inputs and outputs. Thus, if in the defining sequence of a recursive function $f(x)$ the functions $g(x)$ and $h(x,y)$ appear, then terms of the form 'g(5)=7' or 'h(3,2)=10' might appear. Each entry in this sequence needs to be an application of a basic function or follow from the entries above by using composition, primitive recursion or mu recursion. For instance if $f(x) = h(x,g(x))$, then for 'f(5)=3' to appear, terms like 'g(5)=6' and 'h(5,6)=3' must occur above. The computation terminates only if the final term gives the value of the recursive function applied to the inputs.

Markov algorithm
a string rewriting system that uses grammar-like rules to operate on strings of symbols.

Register machine
is a theoretically interesting idealization of a computer. There are several variants. In most of them, each register can hold a natural number (of unlimited size), and the instructions are simple (and few in number), e.g. only decrementation (combined with conditional jump) and incrementation exist (and halting). The lack of the infinite (or dynamically growing) external store (seen at Turing machines) can be understood by replacing its role with Gödel numbering techniques: the fact that each register holds a natural number allows the possibility of representing a complicated thing (e.g. a sequence, or a matrix etc.) by an appropriate huge natural number — unambiguity of both representation and interpretation can be established by number theoretical foundations of these techniques.

P''
Like Turing machines, P'' uses an infinite tape of symbols (without random access), and a rather minimalistic set of instructions. But these instructions are very different, thus, unlike Turing machines, P'' does not need to maintain a distinct state, because all

Church-Turing thesis

- An important statement first made by Alonzo Church and Alan Turing that states that these formal models (Turing machines, unrestricted grammars, lambda calculus, Markov algorithms, etc) are all equally powerful – anything that is intuitively computable can be computed by any of them
- It is just a hypothesis, or conjecture, and cannot be proven because of the vague nature of what is “effectively computable”. Despite this, the thesis is almost universally accepted.

Can these models “compute” everything?

- Are all mathematical functions computable?
- Is it possible to have a generic mathematical theorem prover?
- Is there a grammar for any language?
- Are all well-defined problems solvable?
- Can computers be programmed to do just about anything?

- NO, all these powerful models have their limitations

Turing-decidable and Turing-recognizable languages

- A language is Turing-decidable if there is a TM that can decide if an arbitrary string from Σ^* is a member of L or not within a finite number of steps, i.e. the TM always halts whether the string is accepted or rejected.
- A language L is Turing-acceptable if there is a TM that can check every string in L within a finite number of steps. For strings not in L, the TM may or may not halt.
- Regular $\subset$ CFL $\subset$ Turing-decidable $\subset$ Turing-acceptable
- In some older texts, Turing-decidable = “recursive” and Turing-acceptable = “recursively enumerable”.

The halting problem

- Given an arbitrary Turing machine M and input w for M, will M halt on input w?
- A “universal language” for Turing-recognizable languages

L_U = { M#w : M describes a TM that accepts string w }

- M can describe the TM by listing the states, tape alphabet, and transition rules for the TM
- L_U is Turing-recognizable (by simulating the execution of the TM M on input w)
- However, it can be shown that L_U is not Turing-decidable. *We say that the halting problem is undecidable. A proof?*

Equinumerous sets and the diagonalization method

- Two sets are equinumerous if they have the same number of elements, e.g., { a, b } and { 0, 1 } are equinumerous.
- We can also compare infinite sets, e.g., the set of even numbers and the set of odd numbers are equinumerous. There is a one-to-one and onto function from one set to the other, e.g., f(x) = x+1

  $0 \leftrightarrow 1,\ 2 \leftrightarrow 3,\ 4 \leftrightarrow 5,\ 6 \leftrightarrow 7,\ 8 \leftrightarrow 9,\ \ldots$
- Diagonalization is a method from logic and set theory to prove that certain pairs of sets do not have the same number of elements.

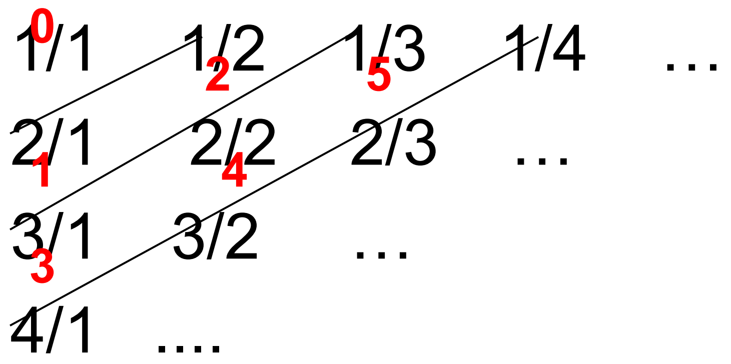
Equinumerous sets

- Surprisingly, the set of natural numbers and the set of even integers are equinumerous – consider the function f(x) = 2x

  $0 \leftrightarrow 0,\ 1 \leftrightarrow 2,\ 2 \leftrightarrow 4,\ 3 \leftrightarrow 6,\ 4 \leftrightarrow 8,\ \ldots$

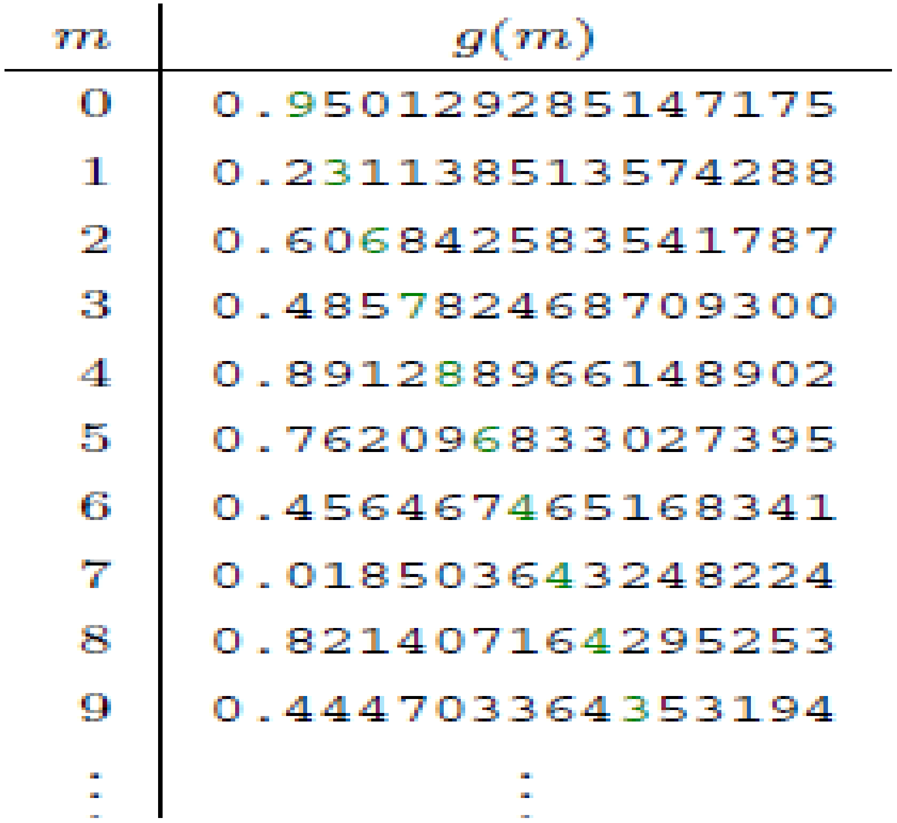
- Note that a set is equinumerous with the set of natural numbers if we can “enumerate” them in some sensible order

- The set of positive rationals is also “enumerable”



Diagonalization

- Set of reals [0.0, 1.0] is <u>not</u> enumerable. Proof is by contradiction using the diagonalization method: Suppose the real numbers can be enumerated... then there is a real number derived from the diagonal digits which differs from every number in the list



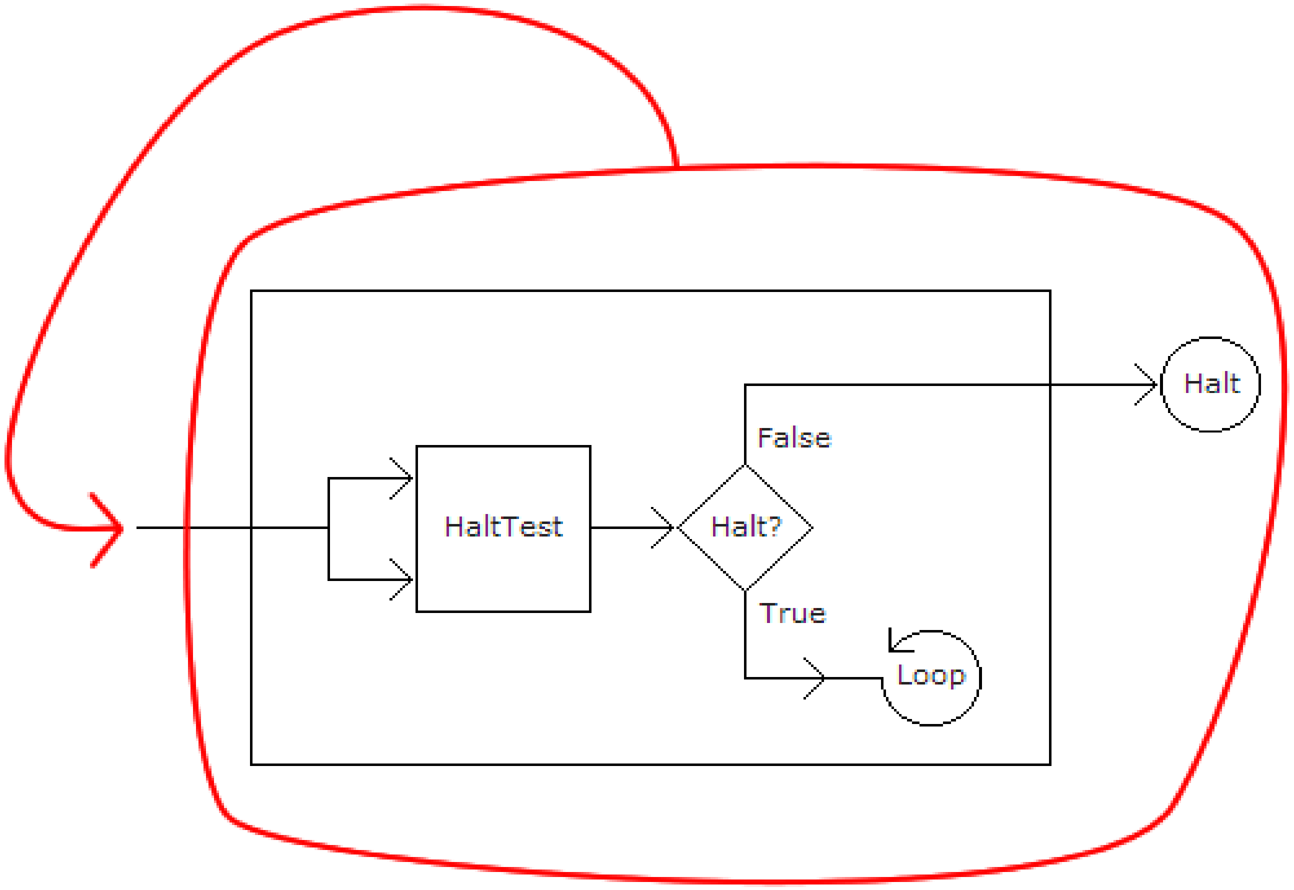
m	$g(m)$
0	0.950129285147175
1	0.231138513574288
2	0.606842583541787
3	0.485782468709300
4	0.891288966148902
5	0.762096833027395
6	0.456467465168341
7	0.018503643248224
8	0.821407164295253
9	0.444703364353194
⋮	⋮

0.8256753332 is not in the list!

this real number is obtained by the taking the jth digit in the jth row and reduced by 1.

A pigeonhole-principle “proof” and a graphical “proof” of the undecidability of the halting problem

- Basically, there are too many problems than there are programs for their solutions --- hence, some problems cannot have solution
-



http://www.ipod.org.uk/reality/reality_strange_feedback.gif

Scooping the Loop Snooper

an elementary proof of the undecidability of the halting problem

No program can say what another will do.
Now, I won't just assert that, I'll prove it to you:
I will prove that although you might work til you drop,
you can't predict whether a program will stop.

Imagine we have a procedure called P
that will snoop in the source code of programs to see
there aren't infinite loops that go round and around;
and P prints the word "Fine!" if no looping is found.

You feed in your code, and the input it needs,
and then P takes them both and it studies and reads
and computes whether things will all end as the should
(as opposed to going loopy the way that they could)...

Rest of poem at http://www.bemuzed.com/lucasd/halting-poem.html