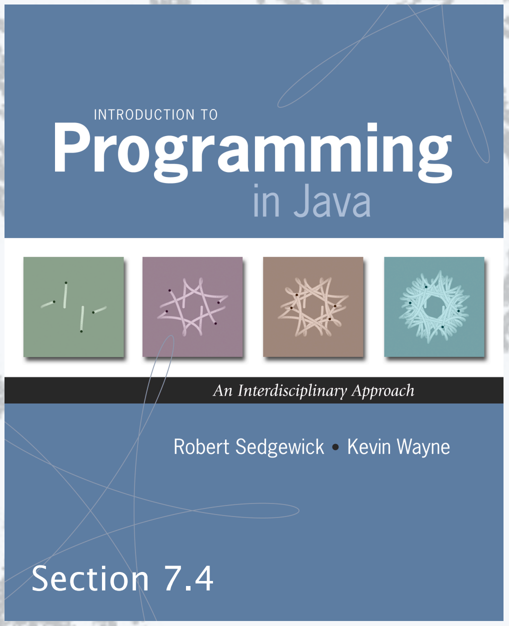
2. Effective calculability.
Abbreviation of treatment. A
function is said to be 'effectively calcu
lable' if its values can be found by some purely
mechanical process. Although it is fairly easy to get
an intuitive grasp of this idea it is nevertheless desirable
to have some definite, mathematically expressible definition.
Such a definition was first given by Gödel at Princeton in 1934
(Gödel [2], 26) following in part an unpublished suggestion of Herb
rand, and has since been developed by Kleene (Kleene [2]). We shall
not be concerned much here with this particular definition. Another defini
tion of effective calculability has been given by Church (Church [3], 356–358)
who identifies it with λ-definability. The author has recently suggested a definition
corresponding more closely to the intuitive idea (Turing [1], see also Post [1]). It was
said above "a function is effectively calculable if its values can be found by some pure
ly mechanical process." We may take this statement literally, understanding by a purely
mechanical process one which could be carried out by a machine. It is possible to give a
mathematical description, in a certain normal form, of the structures of these machines. T
he development of the idea leads to the author's definition of a computable function, and
an identification of computability† with effective calculability. (†We shall use the expressio
n 'computable function' to mean a function calculable by a machine, and let 'effectively c
alculable' refer to the intuitive idea without particular identification with any one of these
definitions. We do not restrict the values taken by a computable function to be natural num
bers; we may for instance have computable propositional functions.) It is not difficult thou
gh somewhat laborious, to prove these three definitions equivalent (Kleene [3], Turing [2]
). In the present paper we shall make considerable use of Church's identification of effec
tive calculability with λ-definability, or, what comes to the same, of the identification w
ith computability and one of the equivalence theorems. In most cases where we have to deal
with an effectively calculable function we shall introduce the corresponding W. F. F. with so
me such phrase as "the function f is effectively calculable, let F be a formula λ-defining it" or
"let F be a formula such that $F(\underline{n})$ is convertible to [illegible] whenever $\underline{n}$ represents a positive integer."
In such cases there is no difficulty in seeing how a machine could in principle be designed to calcu
late the values of the function concerned, and assuming this done the equivalence theorem can be
applied. A statement as to what the formula F actually is may be omitted. We may introduce imm
ediately on this basis a W. F. F. ϖ with the property that ϖ $(\underline{m}, \underline{n})$ conv $\underline{r}$ if r is the greatest positive
integer for which m^r divides n, if any, and is 1 if there is none. We also introduce Dt with the pr
operties Dt $(\underline{n}, \underline{n})$ conv 3; Dt $(\underline{n} + \underline{m}, \underline{n})$ conv 2, Dt $(\underline{n}, \underline{n} + \underline{m})$ conv 1. There is another point to
be made clear in connection with the point of view we are adopting. It is intended that all pr
oofs that are given should be regarded no more critically than proofs in classical analysis.
The subject matter, roughly speaking, is constructive systems of logic, but as the purp
ose is directed towards choosing a particular constructive system of logic for pract
ical use, an attempt at this stage to put our theorems into constructive form w
ould be putting the cart before the horse. Those computable functions whic
h take only the values 0 and 1 are of particular importance since they dete
rmine and are determined by computable properties, as may be seen by r
eplacing '0' and '1' by 'true' and 'false'. But besides this type of proper
ty we may have to consider a different type, which is roughly speaking,
less constructive than the computable properties, but more so than the
general predicates of classical mathematics. Suppose we have a com
putable function of the natural numbers taking natural numbers as values,
then corresponding to this function there is the property of being a value o
f the function. Such a property we shall describe as 'axiomatic'; the reason
for using this term is that it is possible to define such a property by giving a
set of axioms, the property to hold for a given argument if and only if it is p
ossible to deduce that it holds from the axioms. Axiomatic properties may
also be characterized in this way. A property ψ of positive integers is axioma
tic if and only if there is a computable property φ of two positive integers such th
at ψ(x) is true if and only if there is a positive integer y such that φ (x, y) is true.
Or again ψ is axiomatic if and only if there is a W. F. F. F such that ψ (n) is true if
and only if $F(\underline{n})$ conv 2. 3. Number theoretic theorems. By a number theoretic theorem[4]
([4] I believe there is no generally accepted meaning for this term, but it should be noticed that we are using
it in a rather restricted sense. The most generally accepted meaning is probably this: suppose we take an arbitrary
formula of the function calculus of first order and replace the function variables by primitive recursive relations. The resulting formula
represents a typical number theoretic theorem in this [more general] sense.) we shall mean a theorem of the form 'θ (x) vanishes for infin
itely many natural numbers x', where θ (x) is a primitive recursive[5] function. ([5]Primitive recursive functions of natural numbers are
defined inductively as follows.* The class of primitive recursive function is more restricted than the computable functions, but has the advantage
that there is a process whereby one can tell of a set of equations whether it defines a primitive recursive function in the manner described above.
If φ $(x_1, \ldots, x_n)$ is primitive recursive than φ $(x_1, \ldots, x_n) = 0$ is described as a primitive recursive between $x_1, \ldots, x_n$.) We shall say that a problem
is number theoretic if it has been shown that any solution of the problem may be put in the form of a proof of one or more number theoretic
theorems. More accurately we may say that a class of problems is number theoretic if the solution of any one of them can be transformed (by a
uniform process) into the form of proofs of number theoretic theorems. I shall now draw a few consequences from the definitions of 'number
theoretic theorems', and in section 5 will try to justify confining our considerations to this type of problem. An alternative form for number
theoretical theorems is 'for each natural number x there exists a natural number y such that φ (x, y) vanishes', where φ (x, y) is primitive recursive
and conversely. In other words, there is a rule whereby given the function θ (x) we can find a functions φ (x, y), or given ψ (x, y) we can find a
function θ (x), so that 'θ (x) vanishes infinitely often' is a necessary and sufficient condition for 'for each x there is y so that φ $(x, y) = 0$'. In fact given
θ (x) we define φ $(x, y) = θ(y) + α(x, y)$ where α (x, y) is the (primitive recursive) function with the properties α $(x, y) = 1$ $(y < x)$; $= 0$ $(y > x)$. If on the
other hand we are given φ (x, y) we define θ (x) by the equations $θ_1(0) = 3$; $θ_1(x + 1) = 3 \cdot 2^{3}(θ_1(x))^{[illegible]}$ [illegible] $(θ_1(x)) - 1$, $ϖ_2(θ_1(x))$), $θ(x) = φ(ϖ_3$
$(θ_1(x)) - 1$, $ϖ_2(θ_1(x))$) where $ϖ_r(x)$ is to be defined so as to mean 'the largest s for which r^s divides x' and $2^{[illegible]}$ to be defined primitive recursively so
as to have its usual meaning if x is a multiple of 3. The function δ (x) is to be defined by the equations δ $(0) = 0$, δ $(x + 1) = 1$. It is easily verified
that the functions so defined have the desired properties. We shall now show that questions as to the truth of statements of form 'does $f(x)$ vanish
identically', where $f(x)$ is a computable function, can be reduced to questions as to the truth of number theoretical theorems. It is understood that
in each case the rule for the calculation of $f(x)$ is given and that one is satisfied that this rule is valid, i.e. that the machine which should calculate $f(x)$ is
circle free (Turing [1], 232). The function $f(x)$ being computable is general recursive in the Herbrand-Gödel sense, and therefore by a general theorem
due to Kleen[6] ([6]Kleen [3], 727. *Suppose $f(x_1, \ldots, x_{n-1})$, $g(x_1, \ldots, x_n)$, $h(x_1, \ldots, x_{n+1})$ are primitive recursive then φ $(x_1, \ldots, x_n)$ is primitive recursive if it is defined
by one of the sets of equations (a) – (e). (a) φ $(x_1, \ldots, x_n) = h(x_1, \ldots, x_{m-1}, g(x_1, \ldots, x_n), x_{m+1}, \ldots, x_n)$, $(1 \le m \le n)$; (b) φ $(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-1})$; (c) φ $(x_1) = a$,
where $n = 1$ and a is some particular natural number; (d) φ $(x_1) = x_1 + 1$ $(n = 1)$; (e) φ $(x_1, \ldots, x_{n-1}, 0) = f(x_1, \ldots, x_{n-1})$; φ $(x_1, \ldots, x_{n-1}, x_n + 1) = h(x_1, \ldots, x_n, φ(x_1, \ldots, x_n))$. LMC

COMPUTER SCIENCE
SEDGEWICK / WAYNE


INTRODUCTION TO
Programming
in Java
An Interdisciplinary Approach
Robert Sedgewick • Kevin Wayne
Section 7.4

http://introcs.cs.princeton.edu

18. Turing Machines

Universality and computability

Fundamental questions

- What is a general-purpose computer?
- Are there limits on the power of digital computers?
- Are there limits on the power of machines we can build?

Pioneering work at Princeton in the 1930s.

David Hilbert
1862–1943

Asked the questions

Kurt Gödel
1906–1978

Solved the math problem

Alsonzo Church
1903–1995

Solved the decision problem

Alan Turing
1912–1954

Provided THE answers

Context: Mathematics and logic

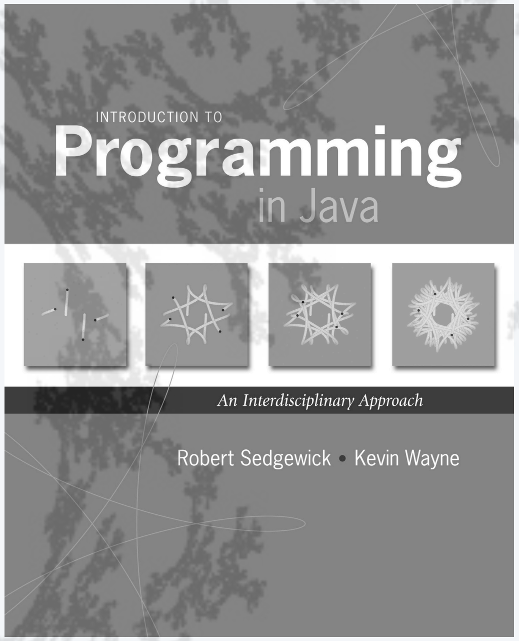
Mathematics. Any formal system powerful enough to express arithmetic.

Principia Mathematics
Peano arithmetic
Zermelo-Fraenkel set theory
.
.
.

Complete. *Can* prove truth or falsity of any arithmetic statement.
Consistent. *Cannot* prove contradictions like $2 + 2 = 5$.
Decidable. An algorithm exists to determine truth of every statement.

Q. (Hilbert, 1900) Is mathematics complete and consistent?
A. (Gödel's Incompleteness Theorem, 1931) NO (!!!)

Q. (Hilbert's Entscheidungsproblem) Is mathematics decidable?
A. (Church 1936, Turing 1936) NO (!!)



http://introcs.cs.princeton.edu

18. Turing Machines

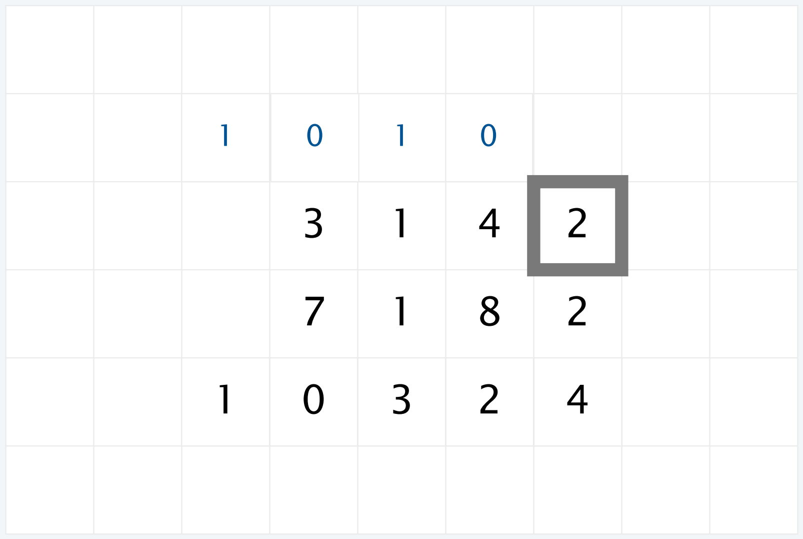
- A simple model of computation
- Universality
- Computability
- Implications

Starting point

Goals

- Develop a model of computation that encompasses all known computational processes.
- Make the model as simple as possible.

Example: A familiar computational process.



Characteristics

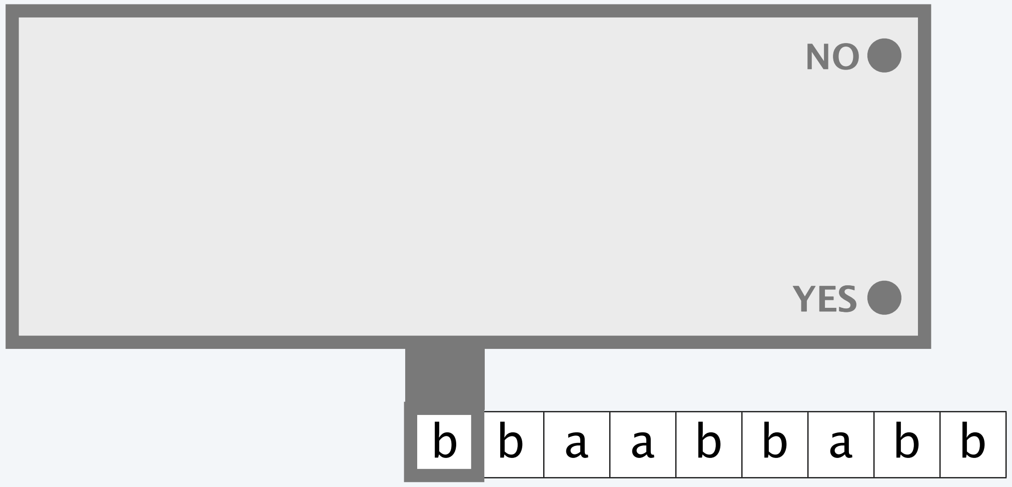
- Discrete.
- Local.
- States.

Previous lecture: DFAs

A DFA is an abstract machine that solves a pattern matching problem.

- A string is specified on an input tape (no limit on its length).
- The DFA reads each character on input tape once, moving left to right.
- The DFA lights "YES" if it *recognizes* the string, "NO" otherwise.

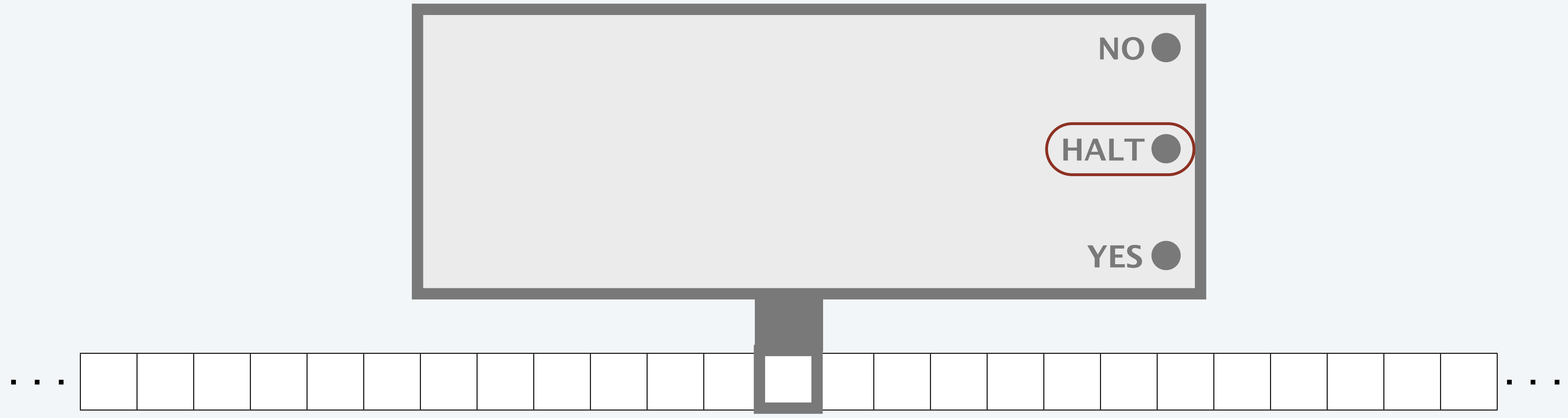
Each DFA defines a *set* of strings (all the strings that it recognizes).



This lecture: Turing machines

A Turing machine (TM) is an abstract model of computation.

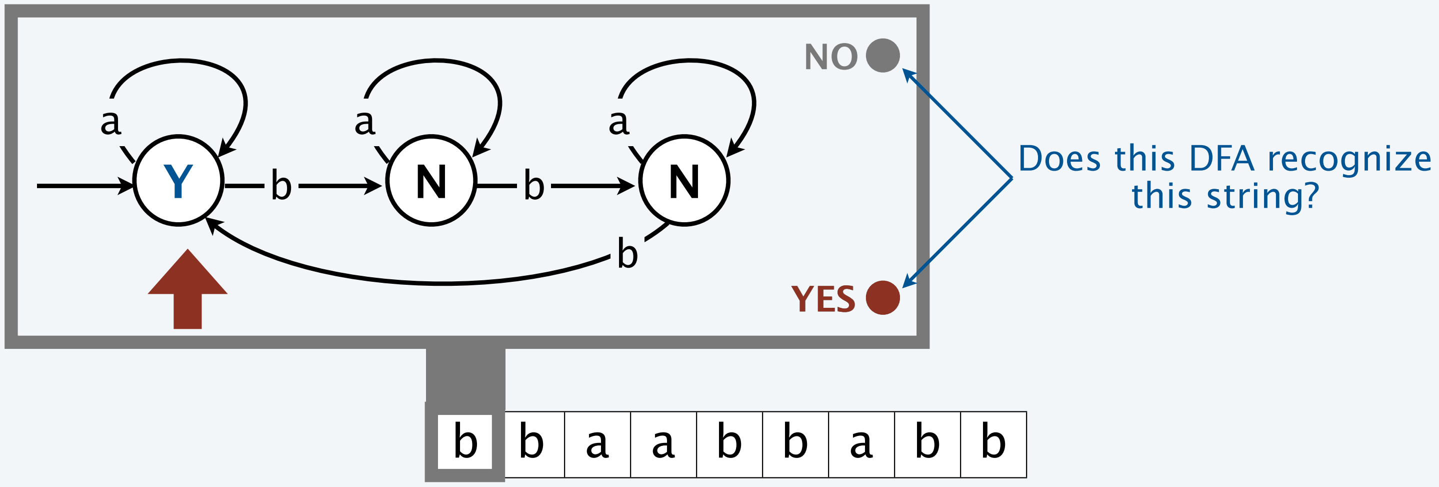
- A string is specified on a tape (no limit on its length).
- The TM reads and writes characters on the tape, moving left or right.
- The TM lights "YES" if it recognizes the string, "NO" otherwise.
- The TM may *halt*, leaving the result of the computation on the tape.



Previous lecture: DFA details and example

A DFA is an abstract machine with a finite number of *states*, each labelled Y or N and *transitions* between states, each labelled with a symbol. One state is the *start* state.

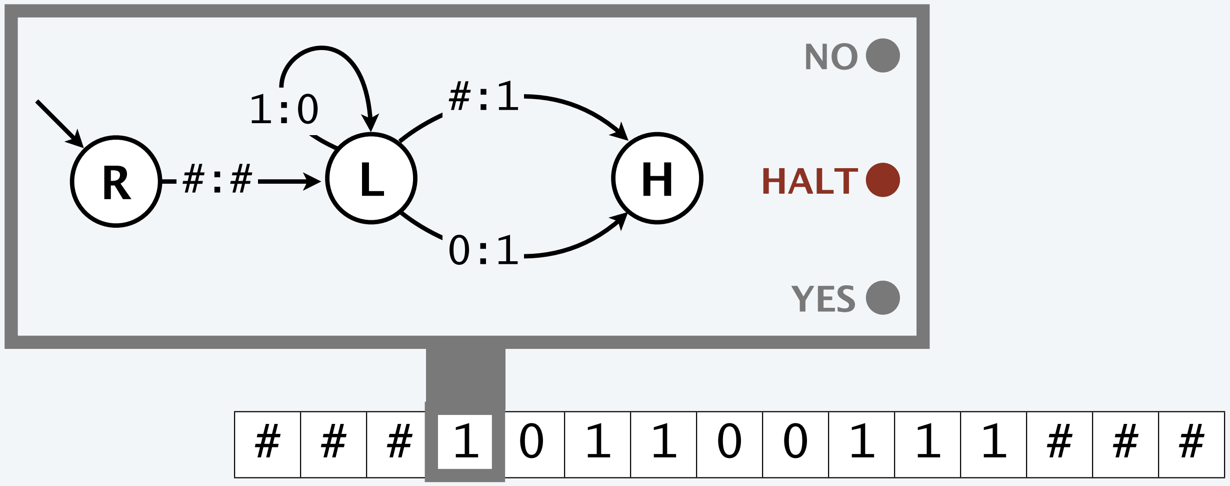
- Begin in the *start* state.
- Read an input symbol and move to the indicated state.
- Repeat until the last input symbol has been read.
- Turn on the "YES" or "NO" light according to the label on the current state.



This lecture: Turing Machine details and example

A Turing Machine is an abstract machine with a finite number of *states*, each labelled Y, N, H, L, or R and *transitions* between states, each labelled with a read/write pair of symbols.

- Begin in the designated *start* state.
- Read an input symbol, move to the indicated state and write the indicated output.
- Move tape head left if new state is labelled L, right if it is labelled R.
- Repeat until entering a state labelled Y, N, or H (and turn on associated light).



DFAs vs TMs

Similarities

- Simple model of computation.
- Input on tape is a finite string with symbols from a finite alphabet.
- Finite number of states.
- State transitions determined by current state and input symbol.

Differences

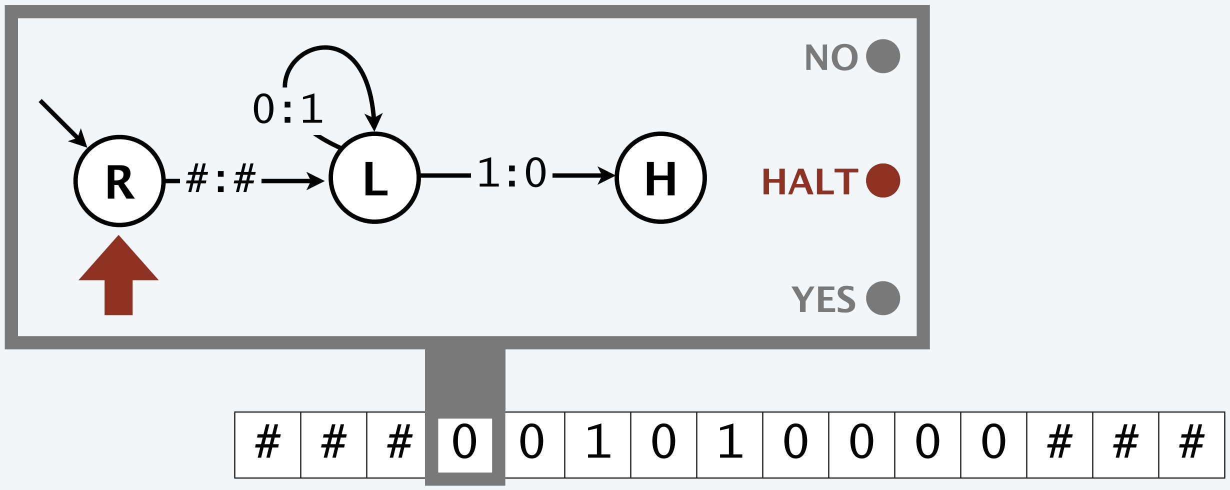
DFAs

- Can read input symbols from the tape.
- Can only move tape head to the right.
- Tape is finite (a string).
- One state transition per input symbol.
- Can *recognize* (turn on "YES" or "NO").

TMs

- Can read from or write onto the tape.
- Can move tape head either direction.
- Tape does not end (either direction).
- No limit on number of transitions.
- Can also *compute* (with output on tape).

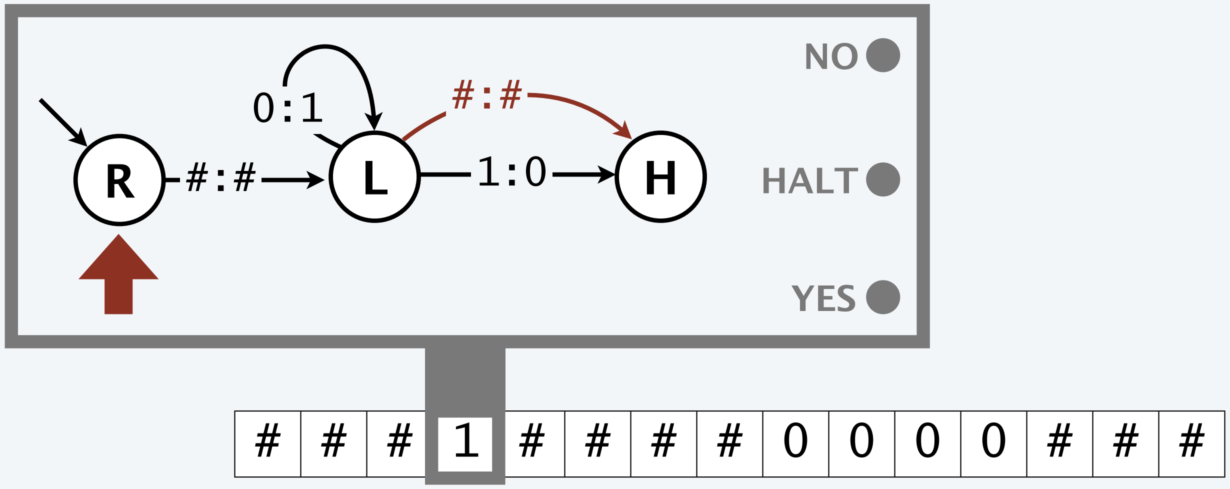
TM example 1: Binary decrementer



Input	1	0	1	0	1	0	0	0	0
Output	1	0	1	0	0	1	1	1	1

TM example 1: Binary decrementer

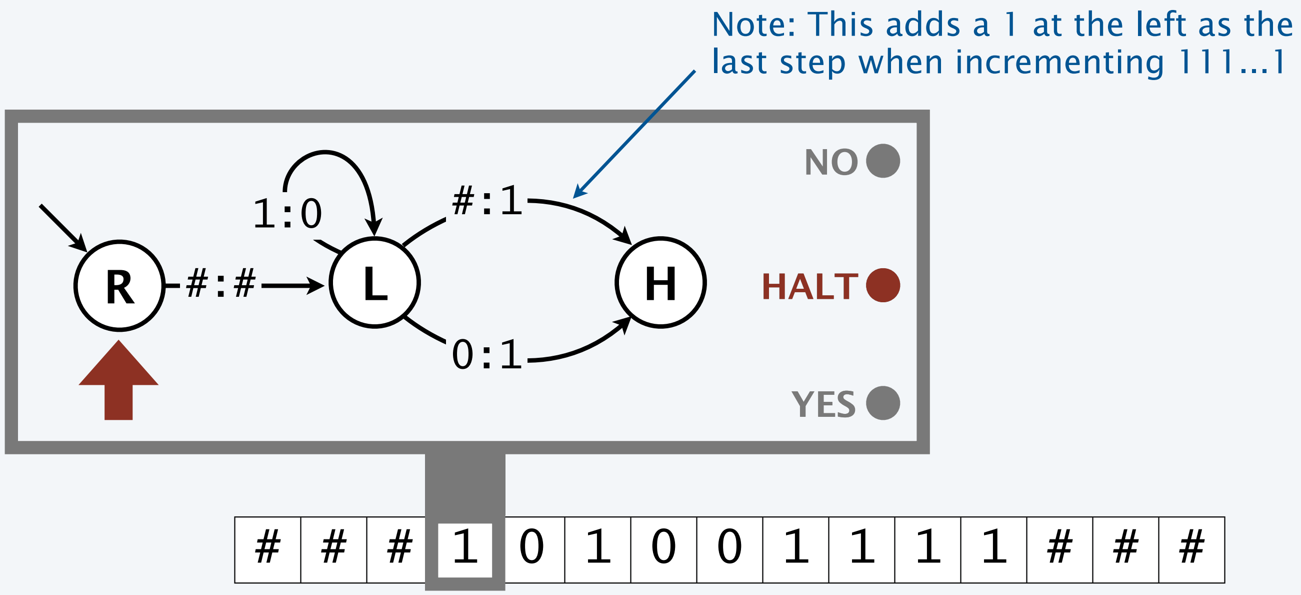
Q. What happens when we try to decrement 0?



A. Doesn't halt! TMs can have bugs, too.

Fix to avoid infinite loop. Check for #.

TM example 2: Binary incrementer

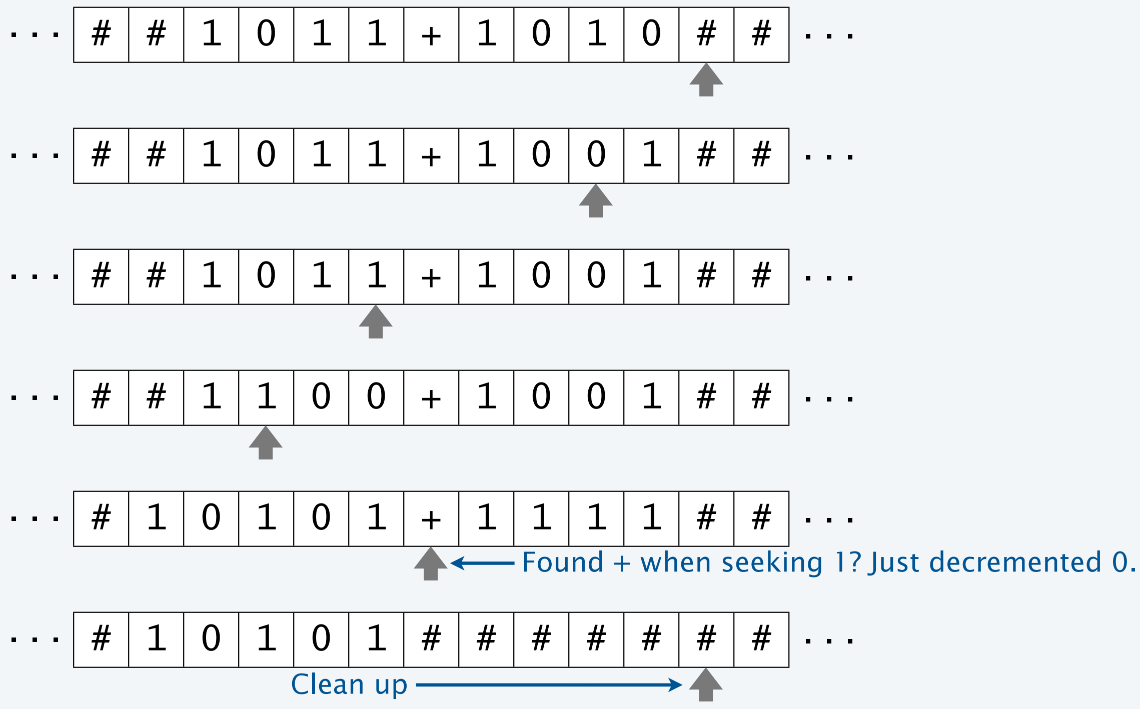

Note: This adds a 1 at the left as the last step when incrementing 111...1
NO
HALT
YES
R
L
H
#:#
1:0
#:1
0:1
# 1 0 1 0 0 1 1 1 1 # #

Input	1	0	1	0	0	1	1	1	1
Output	1	0	1	0	1	0	0	0	0

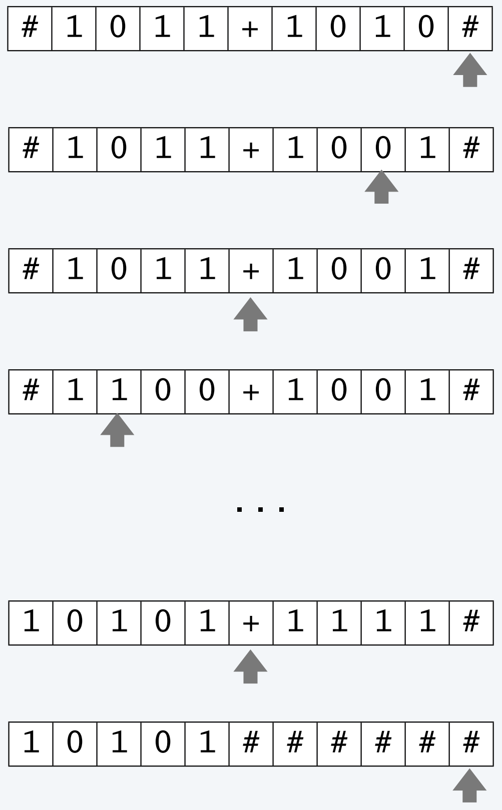
TM example 3: Binary adder (method)

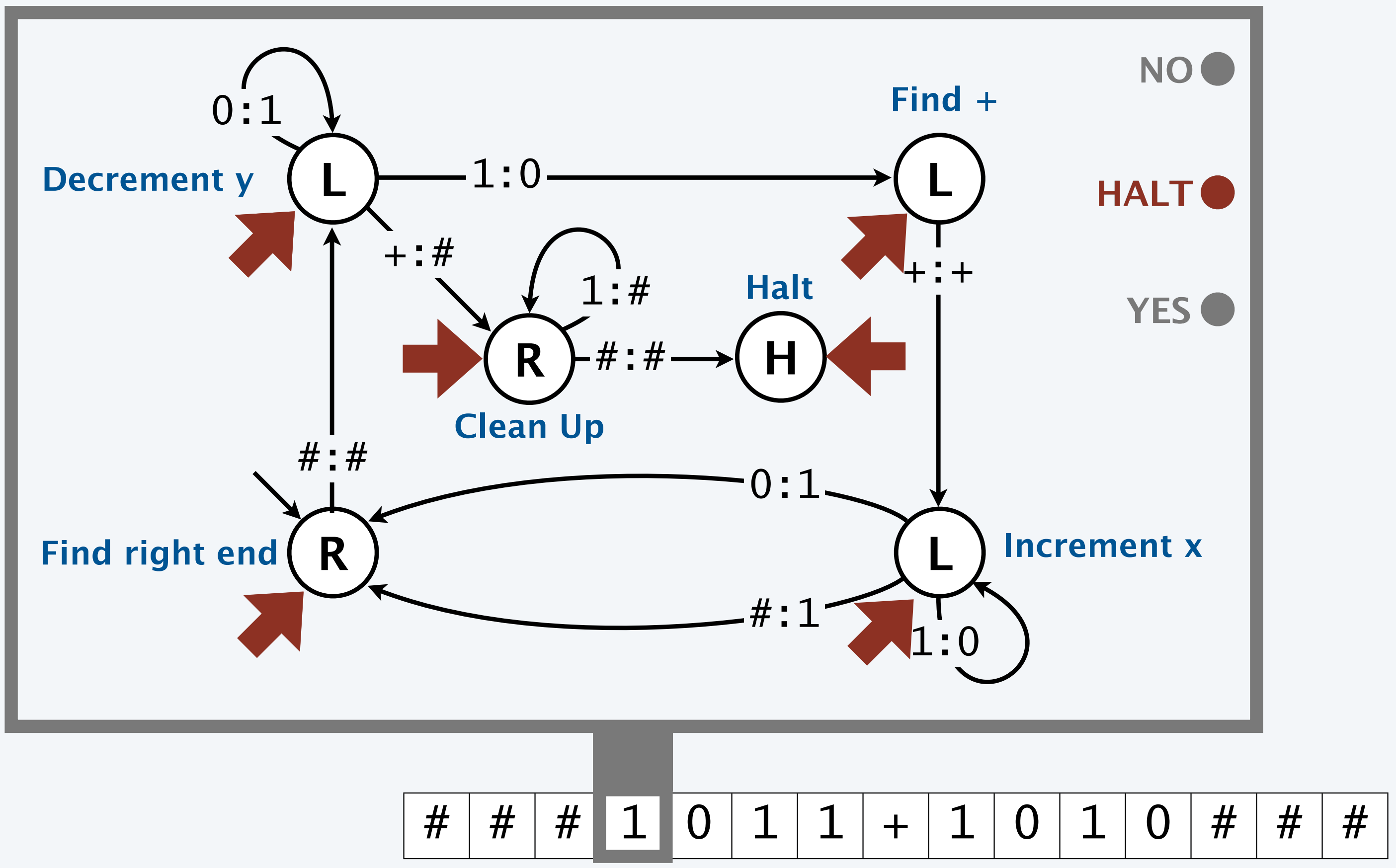
To compute x + y

- Move right to right end of y.
- Decrement y.
- Move left to right end of x (left of +) .
- Increment x.
- Continue until y = 0 is decremented.
- Clean up by erasing + and 1s.



TM example 3: Binary adder





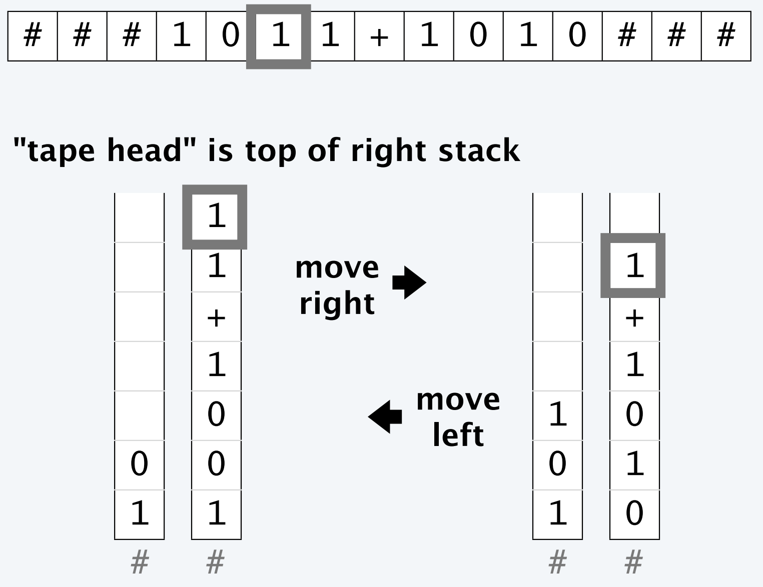
Simulating an infinite tape with two stacks

Q. How can we simulate a tape that is infinite on both ends?

A. Use two stacks, one for each end.

```
private Stack<Character> left;
private Stack<Character> right;

private char read()
{
   if (right.isEmpty()) return '#';
   return right.pop();
}

private char write(char c)
{ right.push(c); }

private void moveRight()
{
   char c = '#';
   if (!right.isEmpty()) c = right.pop();
   left.push(c);
}

private void moveLeft()
{
   char c = '#';
   if (!left.isEmpty()) c = left.pop();
   right.push(c);
}
```

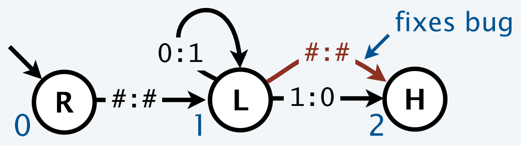
assumes write just after each read



empty? assume # is there

Simulating the operation of a Turing machine

```
public class TM
{
   private int state;
   private int start;
   private String[] action;
   private ST<Character, Integer>[] next;
   private ST<Character, Character>[] out;
   /* Stack code from previous slide */
   public TM(In in)
   {  /* Fill in data structures */  }
   public String simulate(String input)
   {
       state = start;
       for (int i = input.length()-1; i >= 0; i--)
          right.push(input.charAt(i);
       while (action(state).equals("L") ||
                            action(state).equals("R"))
       {
          char c = read();
          state = next[state].get(c);
          write(out[state].get(c));
          if (action[state].equals("R") moveRight();
          if (action[state].equals("L") moveLeft();
       }
       return action[state];
   }
   public static void main(String[] args)
   { /* Similar to DFA's main() */  }
}
```



action[]

0	R
1	L
2	H

next[]

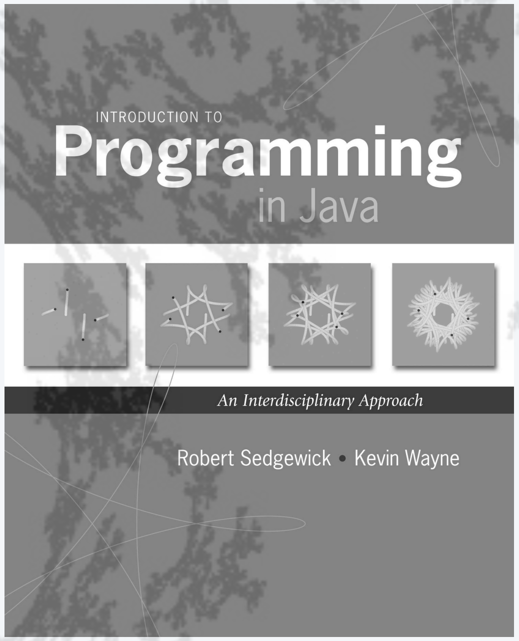
	0	1	#
0	0	0	1
1	1	2	2
2	2	2	2

out[]

	0	1	#
0	0	1	#
1	1	0	#
2	0	1	#

entries in gray are implicit in graphical representation

```
% more dec.txt
3 01# 0
R  0 0 1  0 1 #
L  1 2 2  1 0 #
H  2 2 2  0 1 #

% java TM dec.txt
000111
000110
010000
001111
000000
111111
```

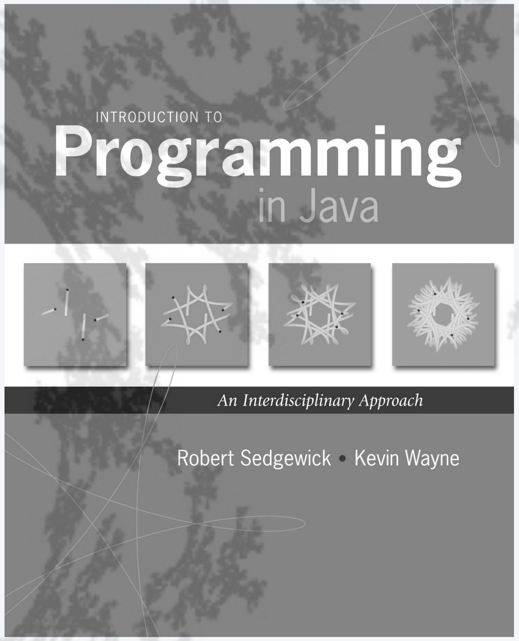


http://introcs.cs.princeton.edu

18. Turing Machines

- **A simple model of computation**
- Universality
- Computability
- Implications

COMPUTER SCIENCE
S E D G E W I C K / W A Y N E



http://introcs.cs.princeton.edu

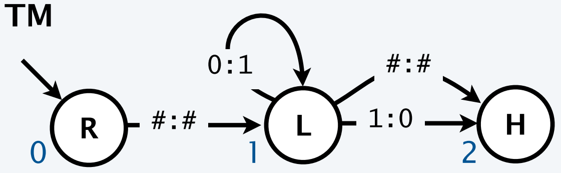
18. Turing Machines

- A simple model of computation
- Universality
- Computability
- Implications

Representing a Turing machine

Turing's key insight. A TM is nothing more than a finite sequence of symbols.

decrementer TM



dec.txt

```
3 01# 0
R   0 0 1   0 1 #
L   1 2 2   1 0 #
H   2 2 2   0 1 #
```

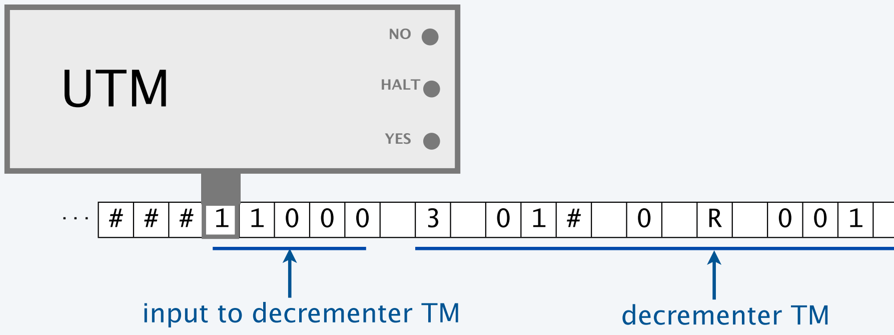
Implication. Can put a TM and its input on a TM tape.

1	1	0	0	0		3		0	1	#		0		R		0	0	1		0	1	#		L		1	2	1		1	0	#		H		2	2	2		0	1	#

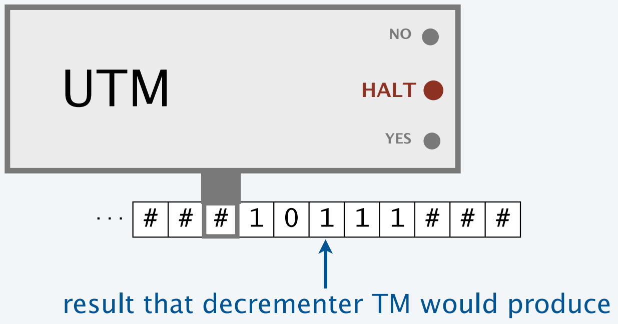
Profound implication. We can use a TM to simulate the operation of any TM.

Universal Turing machine (UTM)

Universal Turing machine. A TM that takes as input any TM and input for that TM on a TM tape.



Result. Whatever would happen if that TM were to run with that input (could loop or end in Y, N or H).



Turing. Simulating a TM is a simple computational task, so there exists a TM to do it: A UTM.

Easier for us to think about. Implement Java simulator as a TM.

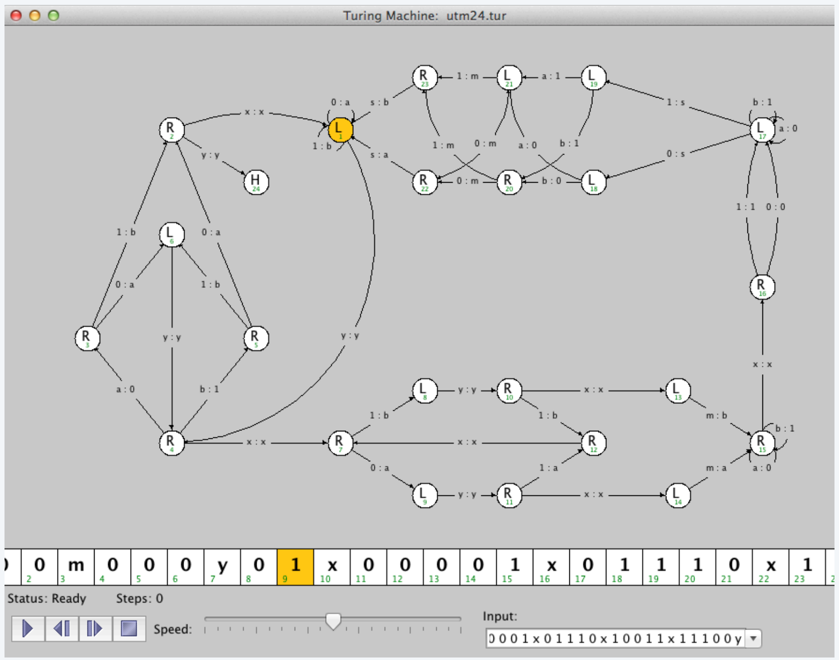
Implementing a universal Turing machine

Java simulator gives a roadmap

- No need for constructor because everything is already on the tape.
- Simulating the infinite tape is a bit easier because TM has an infinite tape.
- Critical part of the calculation is to update `state` as indicated.

Note: This booksite UTM uses a transition-based TM representation that is easier to simulate than the state-based one used in this lecture.

A 24-state UTM



```
0 m 0 0 0 y 0 1 x 0 0 0 0 1 x 0 1 1 1 0 x 1
```

Status: Ready Steps: 0

Input: 0001x01110x10011x11100y

Want to see the details or build your own TM?

Use the booksite's TM development environment.

Warning. TM development may be addictive.

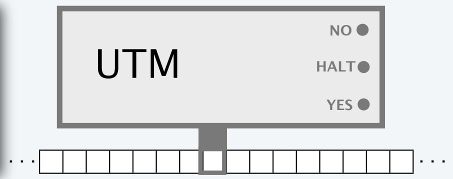
Amazed that it's only 24 states?
The record is 4 states, 6 symbols.

Universality

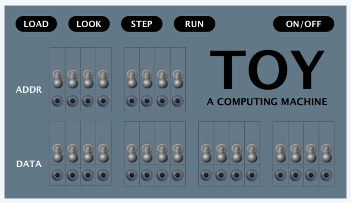
UTM: A *simple* and *universal* model of computation.

Definition. A task is computable if a Turing machine exists that computes it.

Theorem (Turing, 1936). *It is possible to invent a single machine which can be used to do any computable task.*



Profound implications
- Any machine that can simulate a TM can simulate a UTM.
- Any machine that can simulate a TM can do *any* computable task.



A profound connection to the real world

Church-Turing thesis. Turing machines can do anything that can be described by *any* physically harnessable process of this universe: *All computational devices are equivalent*.

Remarks

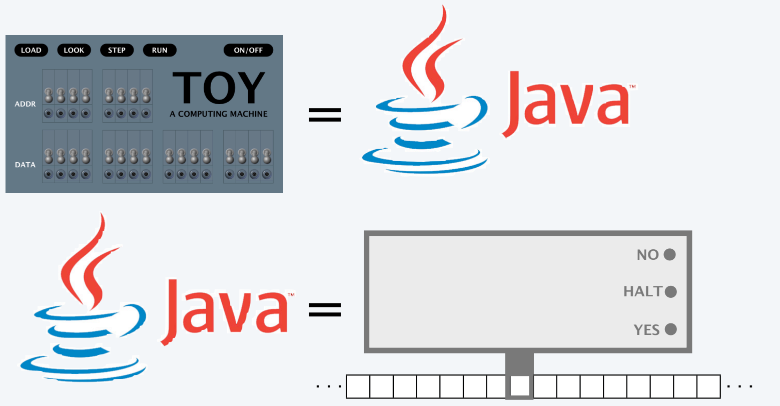
- A thesis, not a theorem.
- *Not* subject to proof.
- *Is* subject to falsification.

New model of computation or new physical process?

- Use *simulation* to prove equivalence.
- Example: TOY simulator in Java.
- Example: Java compiler in TOY.

Implications

- No need to seek more powerful machines or languages.
- Enables rigorous study of computation (in this universe).



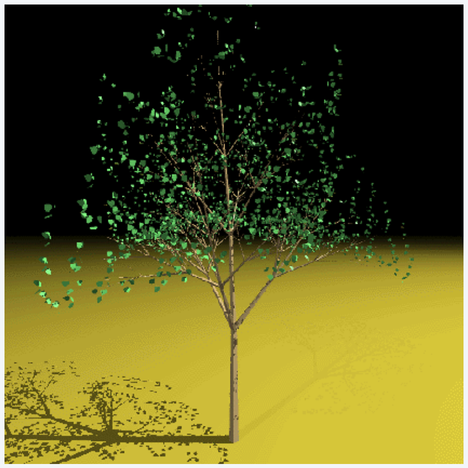
Evidence in favor of the Church-Turing thesis

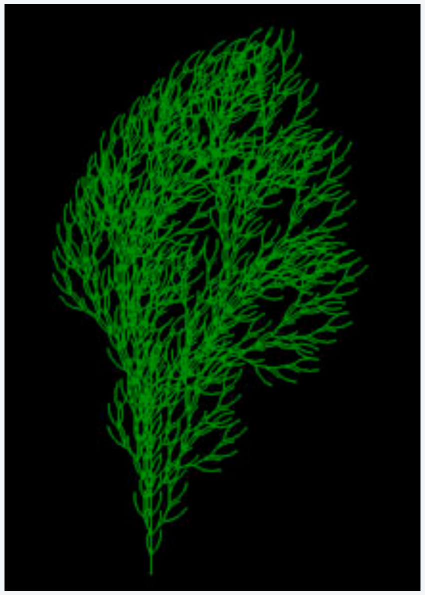
Evidence. Many, many models of computation have turned out to be equivalent (universal).

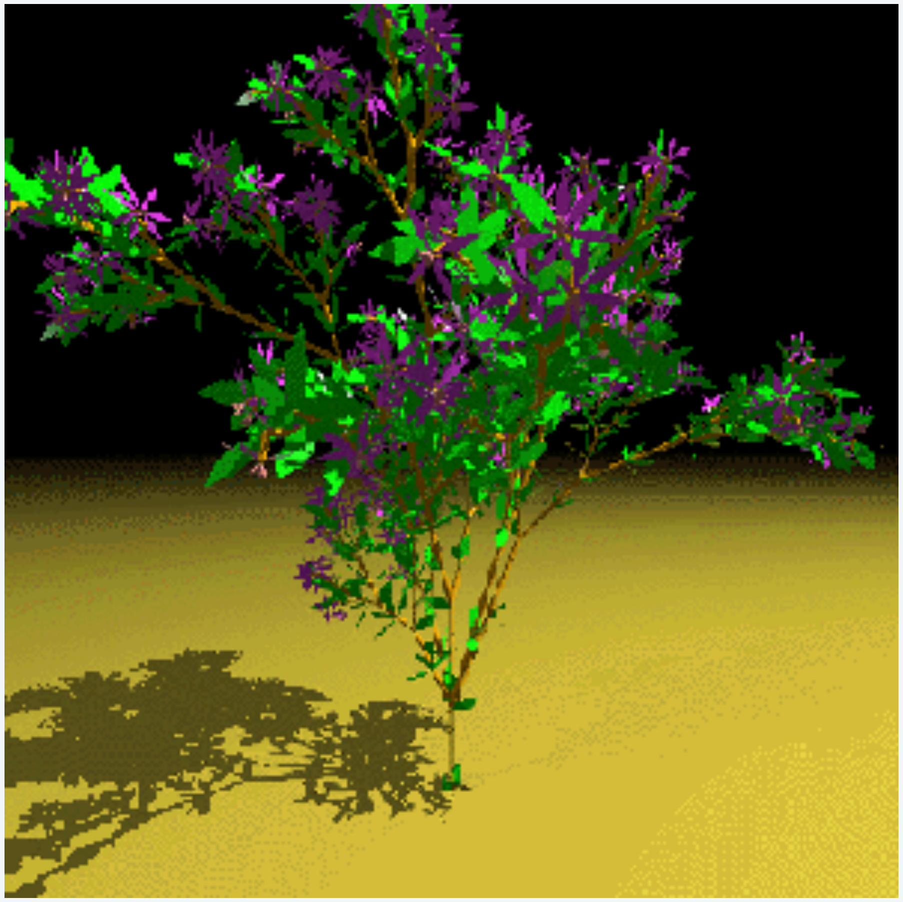
model of computation	*description*
enhanced Turing machines	multiple heads, multiple tapes, 2D tape, nondeterminism
untyped lambda calculus	method to define and manipulate functions
recursive functions	functions dealing with computation on integers
unrestricted grammars	iterative string replacement rules used by linguists
extended Lindenmayer systems	parallel string replacement rules that model plant growth
programming languages	Java, C, C++, Perl, Python, PHP, Lisp, PostScript, Excel
random access machines	registers plus main memory, e.g., TOY, Pentium
cellular automata	cells which change state based on local interactions
quantum computer	compute using superposition of quantum states
DNA computer	compute using biological operations on DNA
PCP systems	string matching puzzles (stay tuned)

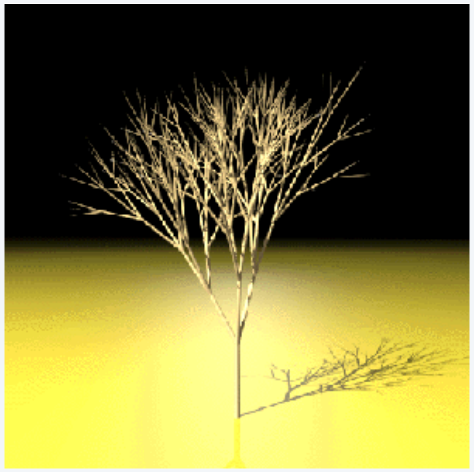
8 decades without a counterexample, and counting.

Example of a universal model: Extended Lindenmayer systems for synthetic plants



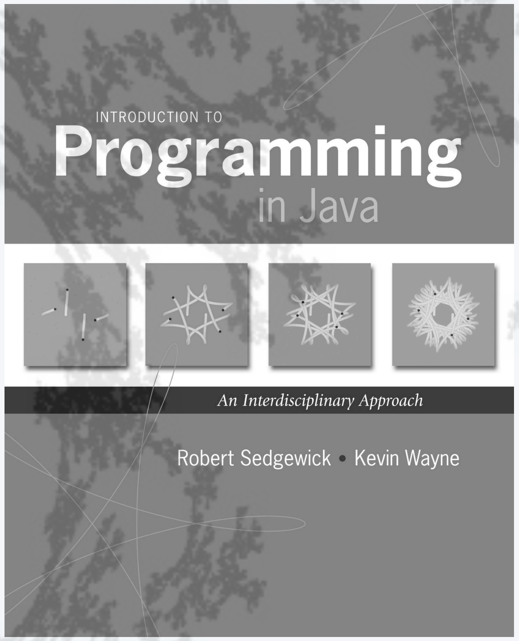






http://astronomy.swin.edu.au/~pbourke/modelling/plants

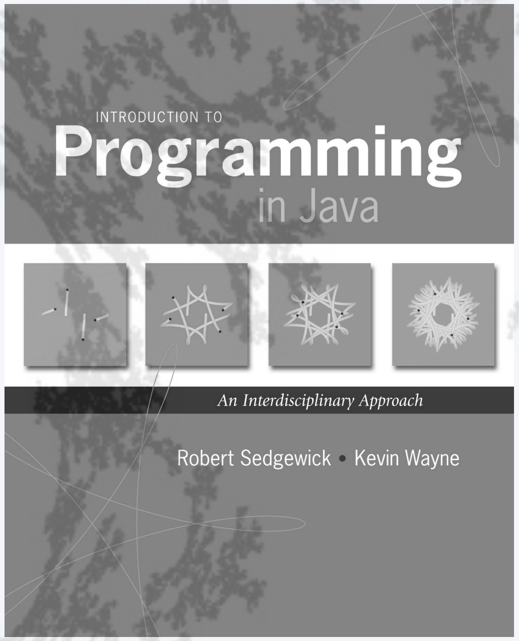
COMPUTER SCIENCE
S E D G E W I C K / W A Y N E



http://introcs.cs.princeton.edu

18. Turing Machines

- A simple model of computation
- Universality
- Computability
- Implications



http://introcs.cs.princeton.edu

18. Turing Machines

- A simple model of computation
- Universality
- Computability
- Implications

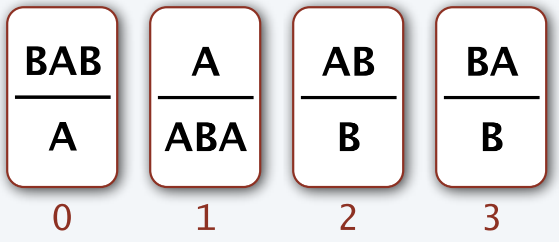
Post's correspondence problem (PCP)

PCP. A family of puzzles, each based on a set of cards.

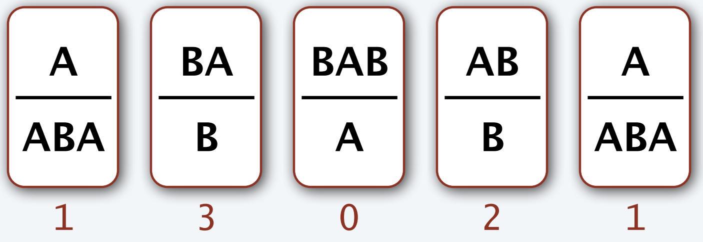
- *N* types of cards.
- No limit on the number of cards of each type.
- Each card has a top string and bottom string.

Does there exist an arrangement of cards with matching top and bottom strings?

Example 1 ($N = 4$).



Solution 1 (easy): YES.



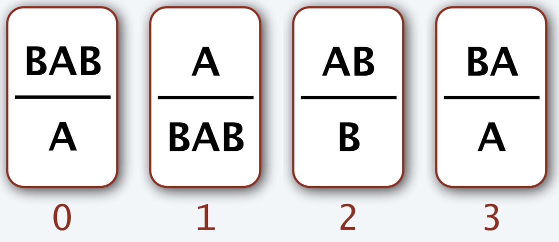
Post's correspondence problem (PCP)

PCP. A family of puzzles, each based on a set of cards.

- *N* types of cards.
- No limit on the number of cards of each type.
- Each card has a top string and bottom string.

Does there exist an arrangement of cards with matching top and bottom strings?

Example 2 ($N = 4$).



Solution 2 (easy): NO. No way to match even the first character!

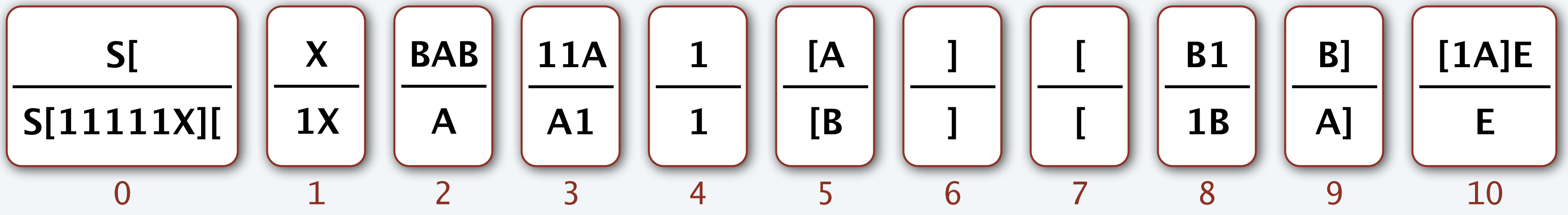
Post's correspondence problem (PCP)

PCP. A family of puzzles, each based on a set of cards.

- *N* types of cards.
- No limit on the number of cards of each type.
- Each card has a top string and bottom string.

Does there exist an arrangement of cards with matching top and bottom strings?

Example 3 (created by Andrew Appel).



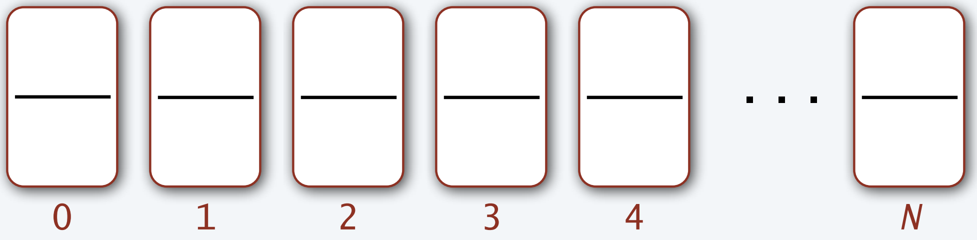
Challenge for the bored: Find a solution that starts with a card of type 0.

Post's correspondence problem (PCP)

PCP. A family of puzzles, each based on a set of cards.

- *N* types of cards.
- No limit on the number of cards of each type.
- Each card has a top string and bottom string.

Does there exist an arrangement of cards with matching top and bottom strings?



A reasonable idea. Write a program to take *N* card types as input and solve PCP.

A surprising fact. It is *not possible* to write such a program.

Another impossible problem

Halting problem. Write a Java program that reads in code for Java static method `f()` and an input `x`, and decides whether or not `f(x)` results in an infinite loop.

Example 1 (easy).

```
public void f(int x)
{
   while (x != 1)
   {
      if  (x % 2 == 0) x = x / 2;
      else             x = 2*x + 1;
   }
}
```

Halts only if x is a positive power of 2

Example 2 (difficulty unknown).

```
public void f(int x)
{
   while (x != 1)
   {
      if  (x % 2 == 0) x = x / 2;
      else             x = 3*x + 1;
   }
}
```

Involves *Collatz conjecture* (see Recursion lecture)

```
 f(7):    7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
f(-17):  -17 -50 -25 -74 -37 -110 -55 -164 -82 -41 -122 ... -17 ...
```

Next. A proof that it is *not possible* to write such a program.

Undecidability of the halting problem

Definition. A yes-no problem is undecidable if no Turing machine exists to solve it.
(A problem is computable if there does exist a Turing machine that solves it.)

Theorem (Turing, 1936). The halting problem is undecidable.

Profound implications

- There exists a problem that no Turing machine can solve.
- There exists a problem that no *computer* can solve.
- There exist *many problems* that no computer can solve (stay tuned).

Warmup: self-referential statements

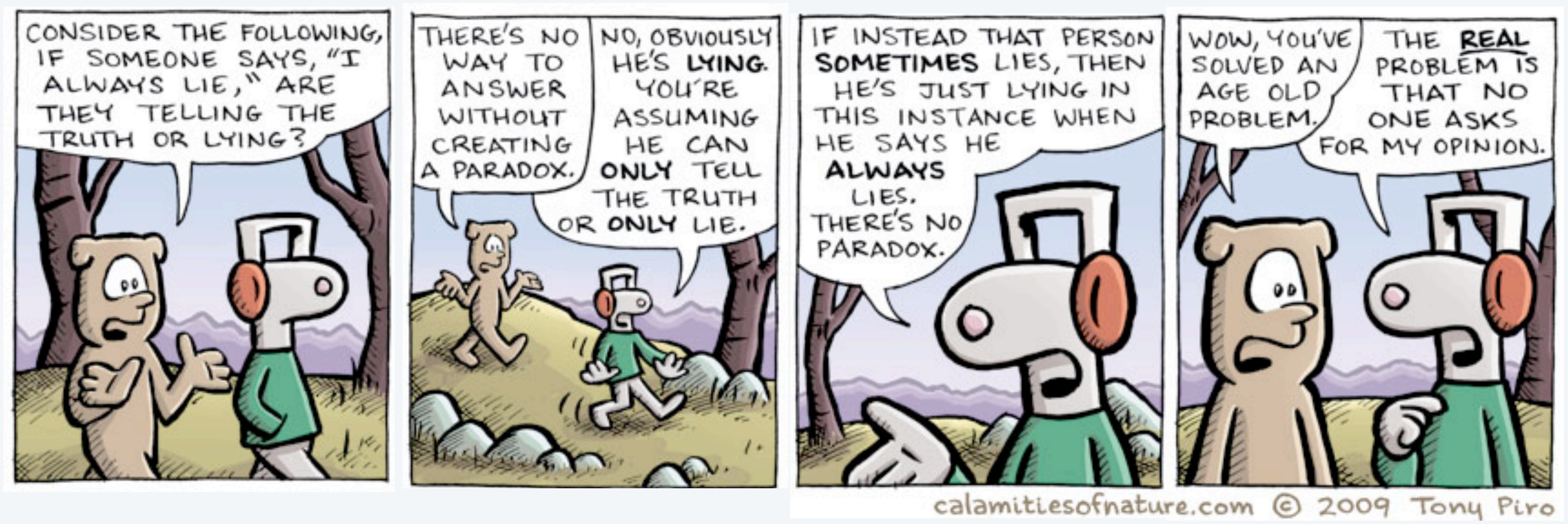
Liar paradox (dates back to ancient Greek philosophers).

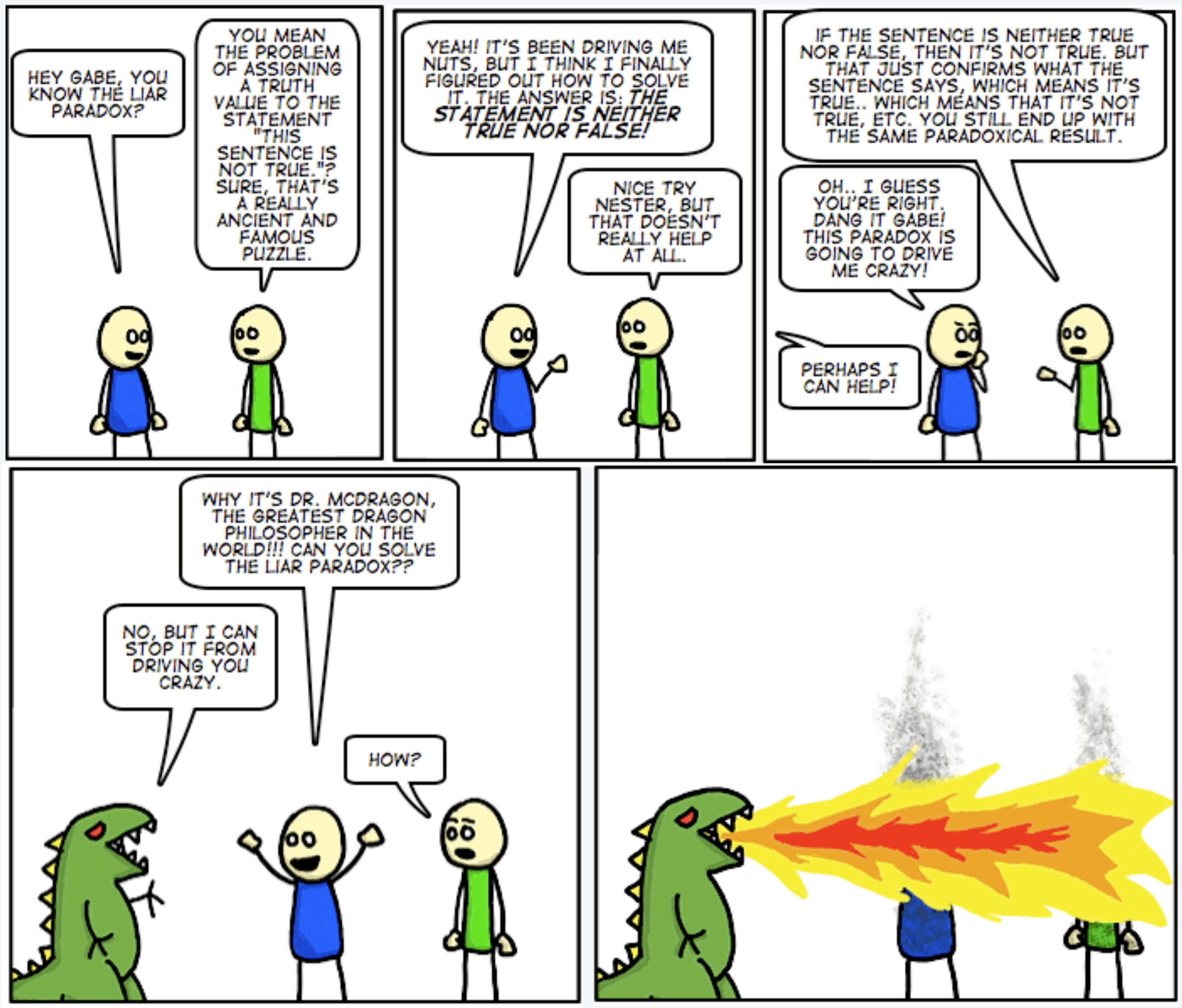
- Divide all statements into two categories: `true` and `false`.
- Consider the statement "This statement is `false`."
- Is it `true`? If so, then it is `false`, a contradiction.
- Is it `false`? If so, then it is `true`, a contradiction.

Logical conclusion. Cannot label *all* statements as `true` or `false`.

Source of the difficulty: Self-reference.

I NEED YOUR SELF-EVALUATION SO I CAN WRITE YOUR PERFORMANCE REVIEW.
REMEMBER TO RATE YOURSELF ON OUR CORE VALUES OF HONESTY AND INTEGRITY.
WALLY CLAIMS HE DID NO WORK THIS YEAR. BUT HE'S DISHONEST, SO YOU CAN'T BE SURE.
www.dilbert.com
scottadams@aol.com
© 2002 United Feature Syndicate, Inc.


CONSIDER THE FOLLOWING, IF SOMEONE SAYS, "I ALWAYS LIE," ARE THEY TELLING THE TRUTH OR LYING?
THERE'S NO WAY TO ANSWER WITHOUT CREATING A PARADOX.
NO, OBVIOUSLY HE'S **LYING**. YOU'RE ASSUMING HE CAN **ONLY** TELL THE TRUTH OR **ONLY** LIE.
IF INSTEAD THAT PERSON **SOMETIMES** LIES, THEN HE'S JUST LYING IN THIS INSTANCE WHEN HE SAYS HE **ALWAYS** LIES. THERE'S NO PARADOX.
WOW, YOU'VE SOLVED AN AGE OLD PROBLEM.
THE **REAL** PROBLEM IS THAT NO ONE ASKS FOR MY OPINION.
calamitiesofnature.com © 2009 Tony Piro


HEY GABE, YOU KNOW THE LIAR PARADOX?
YOU MEAN THE PROBLEM OF ASSIGNING A TRUTH VALUE TO THE STATEMENT "THIS SENTENCE IS NOT TRUE."? SURE, THAT'S A REALLY ANCIENT AND FAMOUS PUZZLE.
YEAH! IT'S BEEN DRIVING ME NUTS, BUT I THINK I FINALLY FIGURED OUT HOW TO SOLVE IT. THE ANSWER IS: *THE STATEMENT IS NEITHER TRUE NOR FALSE!*
NICE TRY NESTER, BUT THAT DOESN'T REALLY HELP AT ALL.
IF THE SENTENCE IS NEITHER TRUE NOR FALSE, THEN IT'S NOT TRUE. BUT THAT JUST CONFIRMS WHAT THE SENTENCE SAYS, WHICH MEANS IT'S TRUE.. WHICH MEANS THAT IT'S NOT TRUE, ETC. YOU STILL END UP WITH THE SAME PARADOXICAL RESULT.
OH.. I GUESS YOU'RE RIGHT. DANG IT GABE! THIS PARADOX IS GOING TO DRIVE ME CRAZY!
PERHAPS I CAN HELP!
WHY IT'S DR. MCDRAGON, THE GREATEST DRAGON PHILOSOPHER IN THE WORLD!!! CAN YOU SOLVE THE LIAR PARADOX??
NO, BUT I CAN STOP IT FROM DRIVING YOU CRAZY.
HOW?

Proof of the undecidability of the halting problem

Theorem (Turing, 1936). The halting problem is undecidable.

Proof outline.

- Assume the existence of a function `halt(f, x)` that solves the problem.

```
public boolean halt(String f, String x)
{
   if ( /* something terribly clever */ ) return true;
   else                                   return false;
}
```

By universality, may as well use Java.
(If this exists, we could simulate it on a TM.)

- Arguments: A function `f` and input `x`, encoded as strings.
- Return value: `true` if `f(x)` halts and `false` if `f(x)` does not halt.
- Always halts.
- Proof idea: *Reductio ad absurdum*: if any logical argument based on an assumption leads to an absurd statement, then the assumption is false.

Proof of the undecidability of the halting problem

Theorem (Turing, 1936). The halting problem is undecidable.

Proof.

- Assume the existence of a function `halt(f, x)` that solves the problem.
- Create a function `strange(f)` that goes into an infinite loop if `f(f)` halts and halts otherwise.
- Call `strange()` with *itself* as argument.
- If `strange(strange)` halts, then `strange(strange)` goes into an infinite loop.
- If `strange(strange)` does not halt, then `strange(strange)` halts.
- *Reductio ad absurdum.*
- `halt(f,x)` cannot exist.

Solution to the problem

```
public boolean halt(String f, String x)
{
   if ( /* f(x) halts */ ) return true;
   else                    return false;
}
```

A client

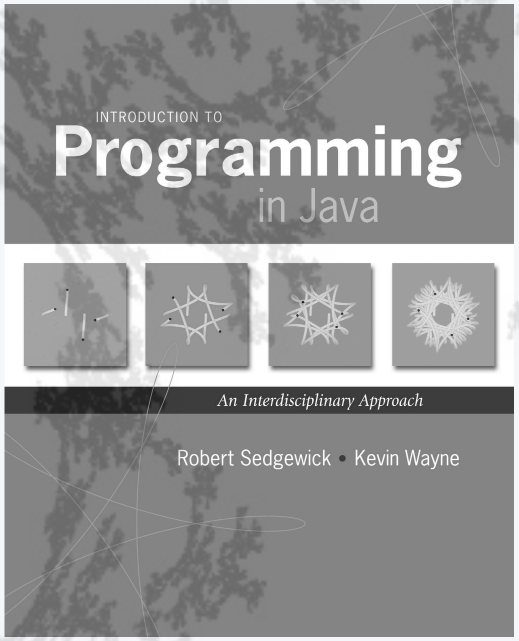
```
public void strange(String f)
{
   if (halt(f, f))
      while (true) { } // infinite loop
}
```

A contradiction

```
strange(strange)
```

halts?
does not halt?

COMPUTER SCIENCE
S E D G E W I C K / W A Y N E

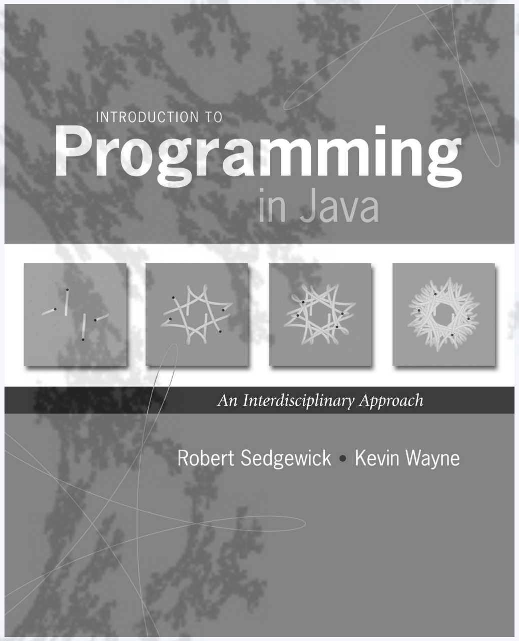


http://introcs.cs.princeton.edu

18. Turing Machines

- A simple model of computation
- Universality
- Computability
- Implications

COMPUTER SCIENCE
SEDGEWICK / WAYNE



http://introcs.cs.princeton.edu

18. Turing Machines

- A simple model of computation
- Universality
- Computability
- Implications

Implications of undecidability

Primary implication. If you know that a problem is undecidable...

Hey, Alice. We came up with a great idea at our hackathon. We're going for startup funding.

What's the idea?

An app that you can use to make sure that any app you download won't hang your phone!

Ummm. I think that's undecidable.

?

Will your app work on *itself*?

???

...don't try to solve it!

Implications for programming systems

Q. Why is debugging difficult?
A. All of the following are *undecidable*.

Halting problem. Give a function f, does it halt on a given input x?
Totality problem. Give a function f, does it halt on *every* input x?
No-input halting problem. Give a function f with no input, does it halt?
Program equivalence. Do two functions f() and g() always return same value?
Uninitialized variables. Is the variable x initialized before it's used?
Dead-code elimination. Does this statement ever get executed?

UNDECIDABLE

Prove each by reduction to the halting problem: A solution would solve the halting problem.

Q. Why are program development environments complicated?
A. They are programs that manipulate programs.

Another undecidable problem

The Entscheidungsproblem (Hilbert, 1928) ← "Decision problem"

- Given a first-order logic with a finite number of additional axioms.
- Is the statement provable from the axioms using the rules of logic?

UNDECIDABLE

David Hilbert
1862–1943

Lambda calculus

- Formulated by Church in the 1930s to address the Entscheidungsproblem.
- Also the basis of modern functional languages.

HASKELL JAVA 8

Alsonso Church
1903–1995

Theorem (Church and Turing, 1936). The Entscheidungsproblem is undecidable.

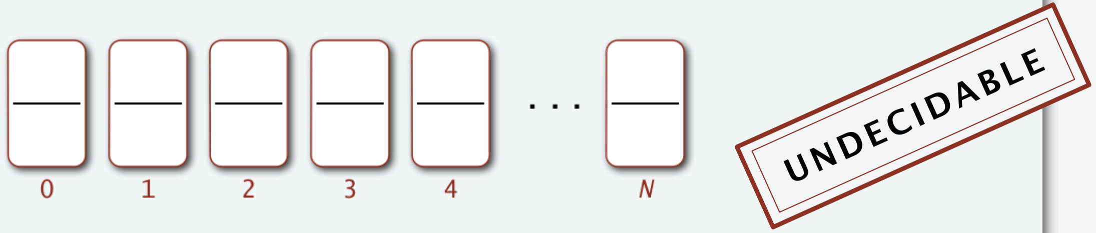
Another undecidable problem

Post's correspondence problem (PCP)

PCP. A family of puzzles, each based on a set of cards.

- *N* types of cards.
- No limit on the number of cards of each type.
- Each card has a top string and bottom string.

Does there exist an arrangement of cards with matching top and bottom strings?



A reasonable idea. Write a program to take *N* card types as input and solve PCP.

Theorem (Post, 1946). Post's correspondence problem is undecidable.

Examples of undecidable problems from computational mathematics

Hilbert's 10th problem

- Given a multivariate polynomial $f(x, y, z, \ldots)$.
- Does f have integral roots ? (Do there exist integers x, y, z, such that $f(x, y, z, \ldots) = 0$?)

UNDECIDABLE

Ex. 1 $f(x,y,z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$

YES $f(5,3,0) = 0$

Ex. 2 $f(x,y) = x^2 + y^2 - 3$ NO

Definite integration

- Given a rational function $f(x)$ composed of polynomial and trigonometric functions.
- Does $\int_{-\infty}^{\infty} f(x)dx$ exist?

UNDECIDABLE

Ex. 1 $\dfrac{\cos(x)}{1+x^2}$ YES $\displaystyle\int_{-\infty}^{\infty} \frac{\cos(x)}{1+x^2}dx = \frac{\pi}{e}$

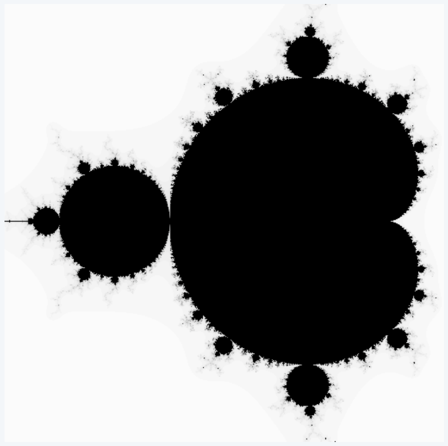
Ex. 2 $\dfrac{\cos(x)}{1-x^2}$ NO

Examples of undecidable problems from computer science

Optimal data compression

- Find the shortest program to produce a given string.
- Find the shortest program to produce a given *picture*.

UNDECIDABLE


produced by a 34-line Java program

Virus identification

- Is this code equivalent to this known virus?
- Does this code contain a virus?

UNDECIDABLE

```
Private Sub AutoOpen()
On Error Resume Next
If System.PrivateProfileString("", CURRENT_USER\Software
\Microsoft\Office\9.0\Word\Security",
                                        "Level") <> "" Then
CommandBars("Macro").Controls("Security...").Enabled = False
. . .
For oo = 1 To AddyBook.AddressEntries.Count
   Peep = AddyBook.AddressEntries(x)
   BreakUmOffASlice.Recipients.Add Peep
   x = x + 1
   If x > 50 Then oo = AddyBook.AddressEntries.Count
Next oo
. . .
BreakUmOffASlice.Subject = "Important Message From " &
Application.UserName
BreakUmOffASlice.Body = "Here is that document you asked
for ... don't show anyone else ;-)"
. . .
```

Melissa virus (1999)

Turing's key ideas

Turing's paper in the *Proceedings of the London Mathematical Society* "On Computable Numbers, With an Application to the Entscheidungsproblem" was one of the most impactful scientific papers of the 20th century.

Alan Turing
1912–1954

The Turing machine. A formal model of computation.

Equivalence of programs and data. Encode both as strings and compute with both.

Universality. Concept of general-purpose programmable computers.

Church-Turing thesis. If it is computable at all, it is computable with a Turing machine.

Computability. There exist inherent limits to computation.

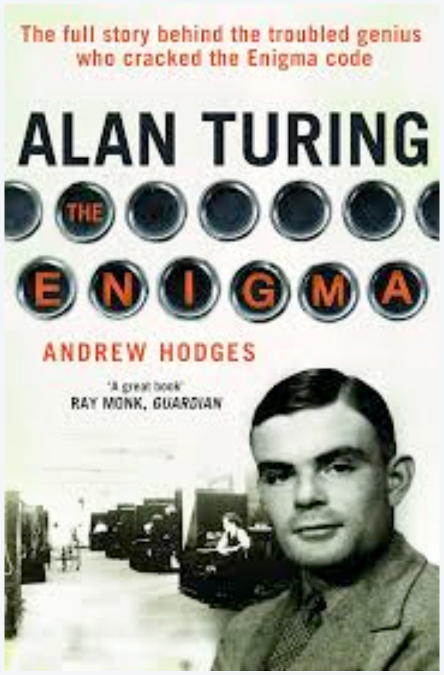
Turing's paper was published in 1936, *ten years before* Eckert and Mauchly worked on ENIAC (!)

John von Neumann read the paper... ← Suggestion: Now go back and read the beginning of the lecture on von Neumann machines

Alan Turing: the father of computer science

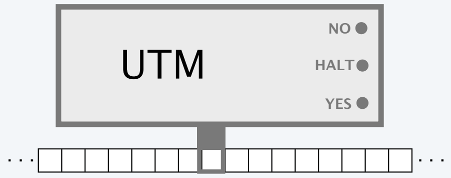
It was not only a matter of abstract mathematics, not only a play of symbols, for it involved thinking about what people did in the physical world…. It was a play of imagination like that of Einstein or von Neumann, doubting the axioms rather than measuring effects…. What he had done was to combine such a naïve mechanistic picture of the mind with the precise logic of pure mathematics. His machines – soon to be called Turing machines – offered ***a bridge, a connection, between abstract symbols and the physical world.***

— *John Hodges, in* Alan Turing, the Enigma

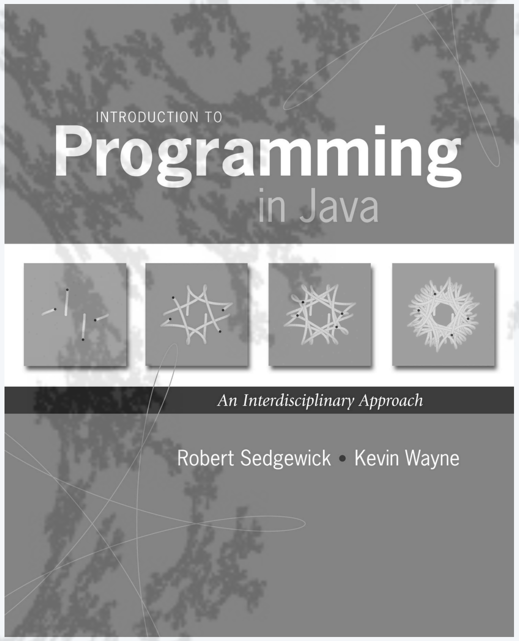

The full story behind the troubled genius who cracked the Enigma code
ALAN TURING
THE
ENIGMA
ANDREW HODGES
'A great book'
RAY MONK, GUARDIAN

A Google data center

=



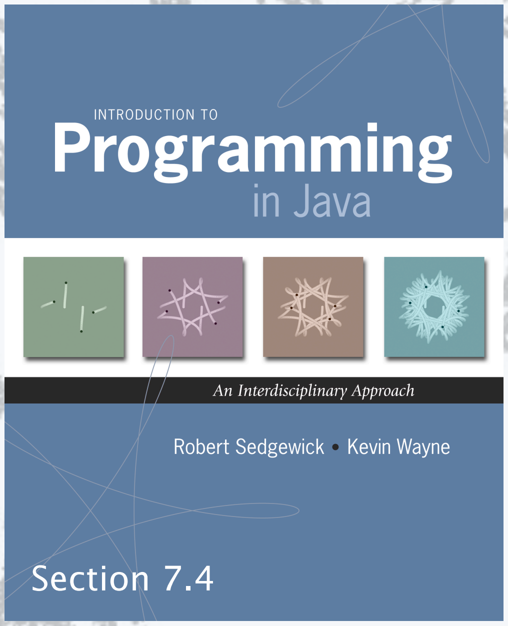
A Universal Turing Machine



http://introcs.cs.princeton.edu

18. Turing Machines

- A simple model of computation
- Universality
- Computability
- Implications



http://introcs.cs.princeton.edu

COMPUTER SCIENCE
SEDGEWICK / WAYNE

18. Turing Machines