CSE 311: Foundations of Computing

Lecture 23 – Undecidability

Last time: Countable sets

A set $\boldsymbol{S}$ is **countable** iff we can order the elements of $\boldsymbol{S}$ as

$$\boldsymbol{S = \{x_1, x_2, x_3, \dots\}}$$

Countable sets:

$\mathbb{N}$ - the natural numbers
$\mathbb{Z}$ - the integers
$\mathbb{Q}$ - the rationals
Σ^* - the strings over any finite Σ
The set of all Java programs

} Shown by "dovetailing"

Last time: Not every set is countable

Theorem [Cantor]:
The set of real numbers between 0 and 1 is not countable.

Proof using "diagonalization".

Uncomputable functions

Interesting… maybe.

Can we come up with an explicit function that is uncomputable?

Uncomputable functions

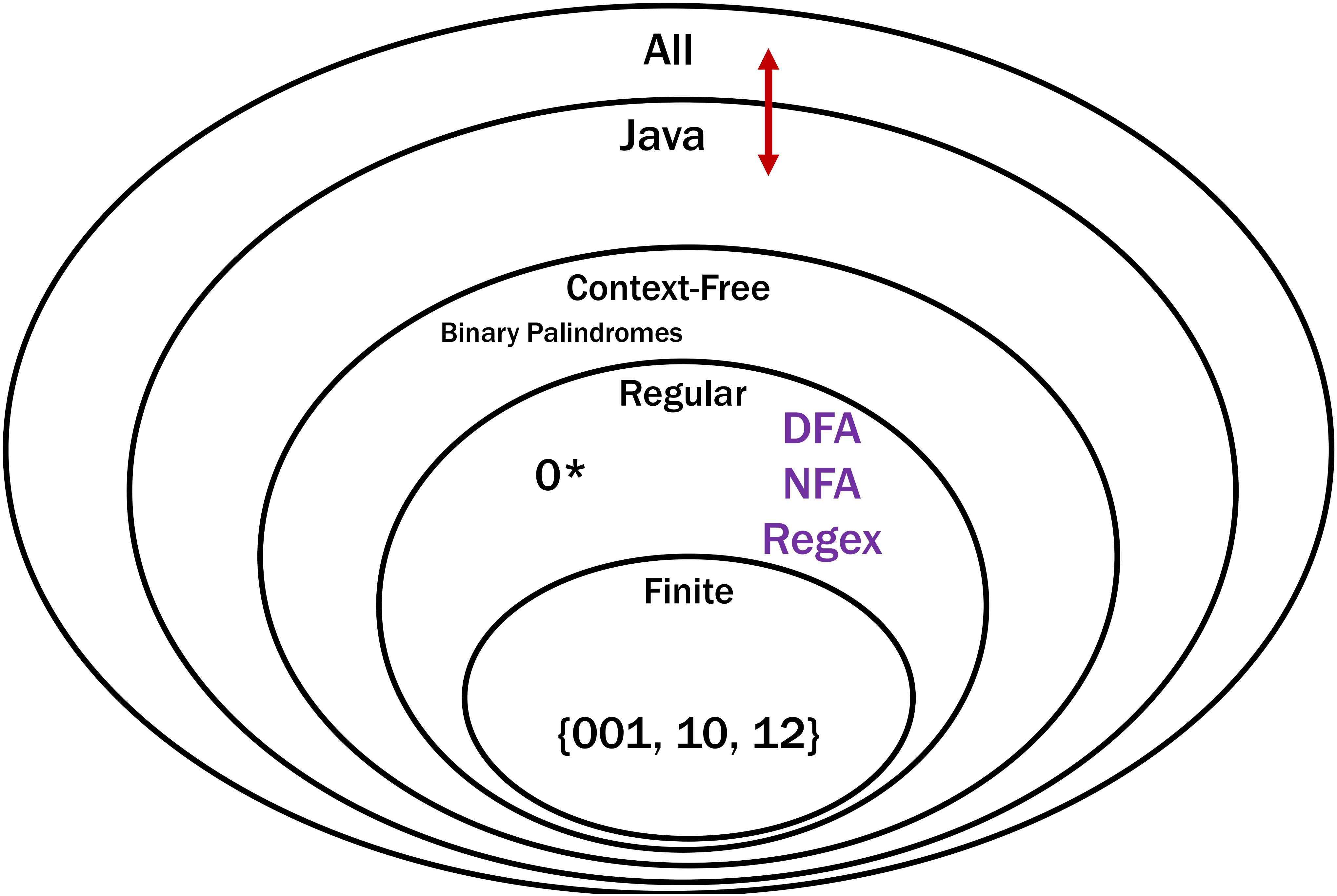
We have seen that:

– The set of all (Java) programs is countable

– The set of all functions $f : \mathbb{N} \rightarrow \{0, \ldots, 9\}$ is not countable

So: There must be some function $f : \mathbb{N} \rightarrow \{0, \ldots, 9\}$ that is not computable by any program!

Recall our language picture



Uncomputable functions

Interesting… maybe.

Can we produce an explicit function that is uncomputable?

A “Simple” Program

```
public static void collatz(n) {
    if (n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return collatz(n/2)
    }
    else {
        return collatz(3*n + 1)
    }
}
```

What does this program do?

… `on n=11?`

… `on n=100000000000000000001?`

11
34
17
52
26
13
40
20
10
5
16
8
4
2
1

A “Simple” Program

```
public static void collatz(n) {
    if (n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return collatz(n/2)
    }
    else {
        return collatz(3*n + 1)
    }
}
```

Nobody knows whether or not this program halts on all inputs!

What does this program do?

… on n=11?

… on n=100000000000000000001?

Some Notation

We're going to be talking about *Java code*.

`CODE(P)` will mean "the code of the program **P**"

So, consider the following function:

```
public String P(String x) {
    return new String(Arrays.sort(x.toCharArray());
}
```

What is **P**(CODE(**P**))?

"(((()))))..;AACPSSaaabceeggghiiiilnnnnnooprrrrrrrrrrssstttttuuwxxyy{}"

Terminology

- **With state machines, we say that a machine "recognizes" the language L iff**
  - **it accepts $x \in \Sigma^*$ if $x \in L$**
  - **it rejects $x \in \Sigma^*$ if $x \notin L$**
- **With Java programs / general computation, we say that the computer "decides" the language L iff**
  - **it halts with output 1 on input $x \in \Sigma^*$ if $x \in L$**
  - **it halts with output 0 on input $x \in \Sigma^*$ if $x \notin L$**

    (difference is the possibility that machine doesn't halt)
- **If no machine decides L, then L is "undecidable"**

The Halting Problem

`CODE(P)` means “the code of the program **P**”

> **The Halting Problem**
>
> **Given:** - CODE(**P**) for any program **P**
> - input **x**
>
> **Output:** **true** if **P** halts on input **x**
> **false** if **P** does not halt on input **x**

Undecidability of the Halting Problem

`CODE(P)` means "the code of the program **P**"

> **The Halting Problem**
>
> **Given:** - CODE(**P**) for any program **P**
> - input **x**
>
> **Output:** **true** if **P** halts on input **x**
> **false** if **P** does not halt on input **x**

Theorem [Turing]: **There is no program that solves the Halting Problem**

Proof by contradiction

Suppose that H is a Java program that solves the Halting problem.

Proof by contradiction

Suppose that H is a Java program that solves the Halting problem.

Then we can write this program:

```
public static void D(String s) {
    if (H(s,s)) {
        while (true);  // don't halt
    } else {
        return;        // halt
    }
}

public static bool H(String s, String x) { ... }
```

Does D(CODE(D)) halt?

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

Does D(CODE(D)) halt?

H solves the halting problem implies that

H(CODE(D),s) is **true** iff D(s) halts, H(CODE(D),s) is **false** iff not

Suppose that D(CODE(D)) **halts**.

Then, by definition of H it must be that

H(CODE(D), CODE(D)) is **true**

Which by the definition of D means D(CODE(D)) **doesn't halt**

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

Does **D**(CODE(**D**)) halt?

H solves the halting problem implies that

  H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

Suppose that **D**(CODE(**D**)) **halts**.

  Then, by definition of **H** it must be that

    H(CODE(**D**), CODE(**D**)) is **true**

  Which by the definition of **D** means **D**(CODE(**D**)) **doesn't halt**

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

H solves the halting problem implies that
H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**),s) is **false** iff not

Suppose that **D**(CODE(**D**)) **halts**.
Then, by definition of **H** it must be that
H(CODE(**D**), CODE(**D**)) is **true**
Which by the definition of **D** means **D**(CODE(**D**)) **doesn't halt**

Suppose that **D**(CODE(**D**)) **doesn't halt**.
Then, by definition of **H** it must be that
H(CODE(**D**), CODE(**D**)) is **false**
Which by the definition of **D** means **D**(CODE(**D**)) **halts**

Does **D**(CODE(**D**)) halt?

```
public static void D(s) {
    if (H(s,s)) {
        while (true);   // don't halt
    } else {
        return;         // halt
    }
}
```

H solves the halting problem implies that

H(CODE(**D**),s) is **true** iff **D**(s) halts, **H**(CODE(**D**) ... ot

Suppose that **D**(CODE(**D**)) **halts.**

Then, by definition of **H** it mus...

H(CODE(**D**), CO...

Which by the defi... **D**(CODE(**D**)) **doesn't halt**

Suppose th... **doesn't halt.**

Th... of **H** it must be that

...ODE(**D**), CODE(**D**)) is **false**

Whi... by the definition of **D** means **D**(CODE(**D**)) **halts**

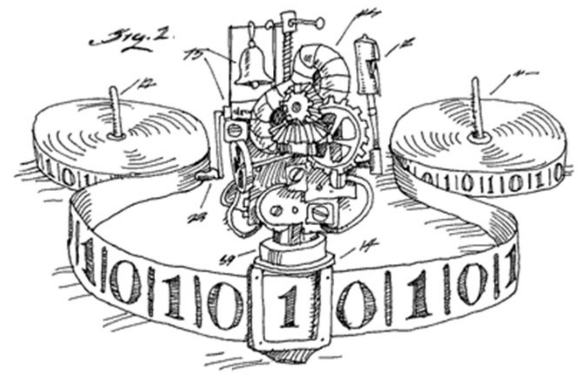
The ONLY assumption was that the program H exists so that assumption must have been false.

Contradiction!

Done

- **We proved that there is no computer program that can solve the Halting Problem.**
  - There was nothing special about Java*

    [Church-Turing thesis]



- This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have.

Where did the idea for creating D come from?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true);  // don't halt
    } else {
        return;        // halt
    }
}
```

D halts on input code(P) iff H(code(P),code(P)) outputs false
iff P doesn't halt on input code(P)

Connection to diagonalization

Write <P> for CODE(P)

Some possible inputs **x**

	<P_1>	<P_2>	<P_3>	<P_4>	<P_5>	<P_6>	....
P_1							
P_2							
P_3							
P_4							
P_5							
P_6							
P_7							
P_8							
P_9							
.							
.							

All programs **P**

This listing of all programs really does exist since the set of all Java programs is countable

The goal of this "diagonal" argument is not to show that the listing is incomplete but rather to show that a "flipped" diagonal element is not in the listing

Connection to diagonalization

Write $<P>$ for CODE(P)

Some possible inputs **x**

All programs **P**

	$<P_1>$	$<P_2>$	$<P_3>$	$<P_4>$	$<P_5>$	$<P_6>$	....					
P_1	0	1	1	0	1	1	1	0	0	0	1	...
P_2	1	1	0	1	0	1	1	0	1	1	1	...
P_3	1	0	1	0	0	0	0	0	0	0	1	...
P_4	0	1	1	0	1	0	1	1	0	1	0	...
P_5	0	1	1	1	1	1	1	0	0	0	1	...
P_6	1	1	0	0	0	1	1	0	1	1	1	...
P_7	1	0	1	1	0	0	0	0	0	0	1	...
P_8	0	1	1	1	1	0	1	1	0	1	0	...
P_9	. .	. .	.	. .	.	.	.	.		.		
.	. .	. .	.	. .	.	.	.	.		.		
.												

(P,x) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

Connection to diagonalization

Write <P> for CODE(P)

Some possible inputs **x**

Want behavior of program ***D*** to be like the flipped diagonal, so it can't be in the list of all programs.

All programs **P**

	<P_1>	<P_2>	<P_3>	<P_4>	<P_5>	<P_6>	....					
P_1	0^{1}	1	1	0	1							
P_2	1	1^{0}	0	1	0							
P_3	1	0	1^{0}	0	0							
P_4	0	1	1	0^{1}	1	0	1	1	0	1	0	...
P_5	0	1	1	1	1^{0}	1	1	0	0	0	1	...
P_6	1	1	0	0	0	1^{0}	1	0	1	1	1	...
P_7	1	0	1	1	0	0	0^{1}	0	0	0	1	...
P_8	0	1	1	1	1	0	1	1^{0}	0	1	0	...
P_9	.	.	. .	.	.	.	.	.	.	.		.
.	.	.	. .	.	.	.	.	.	.	.		.
.												

(P,x) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

Where did the idea for creating D come from?

```
public static void D(s) {
    if (H(s,s) == true) {
        while (true); /* don't halt */
    }
    else {
        return;       /*   halt   */
    }
}
```

D halts on input code(P) iff H(code(P),code(P)) outputs false
iff P doesn't halt on input code(P)

Therefore, for any program P, D differs from P on input code(P)

The Halting Problem isn't the only hard problem

- Can use the fact that the Halting Problem is undecidable to show that other problems are undecidable

General method (a "reduction"):

Prove that, if there were a program deciding B, then there would be a program deciding the Halting Problem.

"B decidable $\rightarrow$ Halting Problem decidable"

Contrapositive:

"Halting Problem undecidable $\rightarrow$ B undecidable"

Therefore, B is undecidable

A CSE 142 assignment

Students should write a Java program that:

- Prints "Hello" to the console
- Eventually exits

GradeIt, PracticeIt, etc. need to grade these

How do we write that grading program?

WE CAN'T: THIS IS IMPOSSIBLE!

Another undecidable problem

- **CSE 142 Grading problem:**
  - **Input: CODE(Q)**
  - **Output:**

    True if **Q** outputs "HELLO" and exits

    False if **Q** does not do that

- **Theorem: The CSE 142 Grading is undecidable.**
- **Proof idea: Show that, if there is a program T to decide CSE 142 grading, then there is a program H to decide the Halting Problem for code(P) and input x.**

Another undecidable problem

Theorem: The CSE 142 Grading is undecidable.

Proof: Suppose there is a program T that decide CSE 142 grading problem. Then, there is a program H to decide the Halting Problem for code(P) and input x by

- transform P (with input x) into the following program Q

Another undecidable problem

```
public class Q {
  private static String x = "...";

  public static void main(String[] args) {
    PrintStream out = System.out;
    System.setOut(new PrintStream(
        new WriterOutputStream(new StringWriter())));
    System.setIn(new ReaderInputStream(new StringReader(x)));

    P.main(args);

    out.println("HELLO");
  }
}

class P {
  public static void main(String[] args) { ... }
  ...
}
```

Another undecidable problem

Theorem: The CSE 142 Grading is undecidable.

Proof: Suppose there is a program T that decide CSE 142 grading problem. Then, there is a program H to decide the Halting Problem for code(P) and input x by

- **transform P (with input x) into the following program Q**
- **run T on code(Q)**
  - **if it returns true, then P halted**

    must halt in order to print "HELLO"
  - **if it returns false, then P did not halt**

    program Q can't output anything other than "HELLO"

More Reductions

- Can use undecidability of these problems to show that other problems are undecidable.

- For instance:

$\mathrm{EQUIV}(P, Q)$:

True	if $P(x)$ and $Q(x)$ have the same behavior for every input x
False	otherwise

Rice's theorem

Not *every* problem on programs is undecidable!

Which of these is decidable?

- Input `CODE(P)` and x
  Output: true if P prints "ERROR" on input x
  after less than 100 steps
  false otherwise

- Input `CODE(P)` and x
  Output: true if P prints "ERROR" on input x
  after more than 100 steps
  false otherwise

Rice's Theorem:

Any "non-trivial" property of the input-output behavior of Java programs is undecidable.

Rice's theorem

Not *every* problem on programs is undecidable!

Which of these is decidable?

- Input `CODE(P)` and x
  Output: true if P prints "ERROR" on input x
  after less than 100 steps
  false otherwise

- Input `CODE(P)` and x
  Output: true if P prints "ERROR" on input x
  after more than 100 steps
  false otherwise

Rice's Theorem (a.k.a. Compilers ARE DIFFICULT):
Any "non-trivial" property of the input-output behavior of Java programs is undecidable.

CFGs are complicated

We know can answer almost any question about REs

- Do two RegExps recognize the same language?

But many problems about CFGs are undecidable!

- **Do two CFGs generate the same language?**
- **Is there any string that two CFGs both generate?**
  - **more general: “CFG intersection” problem**
- **Does a CFG generate *every* string?**

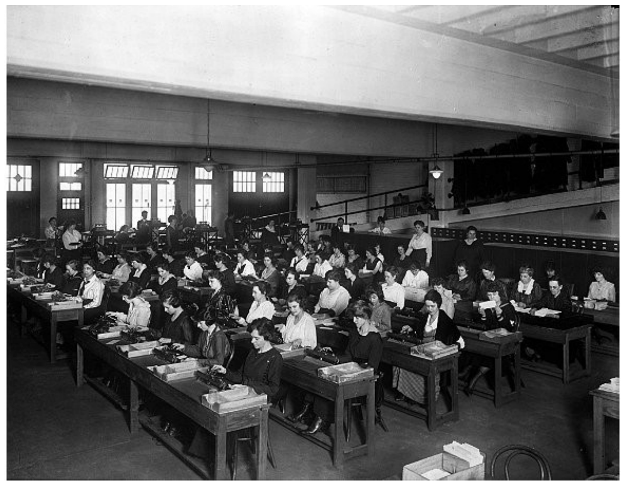
Takeaway from undecidability

- You can't rely on the idea of improved compilers and programming languages to eliminate all programming errors
  - truly safe languages can't possibly do general computation
- Document your code
  - there is no way you can expect someone else to figure out what your program does with just your code; since in general it is provably impossible to do this!

Computers and algorithms

- Does Java (or any programming language) cover all possible computation? Every possible algorithm?

- There was a time when computers were people who did calculations on sheets paper to solve computational problems



- Computers as we known them arose from trying to understand everything these people could do.

Before Java

1930's:

How can we formalize what algorithms are possible?

- **Turing machines** (Turing, Post)
  – basis of modern computers
- **Lambda Calculus** (Church)
  – basis for functional programming, LISP
- **μ-recursive functions** (Kleene)
  – alternative functional programming basis

Turing machines

Church-Turing Thesis:

Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

Evidence

- Huge numbers of models based on radically different ideas turned out to be equivalent to TMs
- TM can simulate the physics of any machine that we could build (even quantum computers)

Turing machines

- **Finite Control**
  - **Brain/CPU** that has only a **finite # of possible "states of mind"**
- **Recording medium**
  - **An unlimited supply of** blank **"scratch paper"** on which to **write & read symbols,** each chosen from a **finite set of possibilities**
  - **Input** also supplied **on** the **scratch paper**
- **Focus of attention**
  - **Finite control** can only **focus on** a **small portion of the recording medium at once**
  - **Focus of attention** can only **shift a small amount at a time**

Turing machines

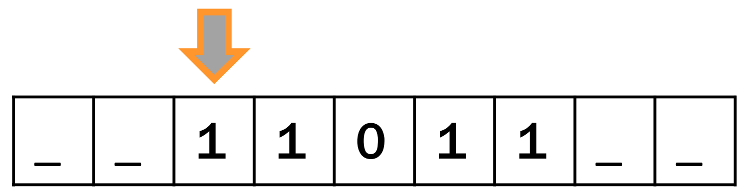
- **Recording medium**
  - An **infinite** read/write "**tape**" marked off into **cells**
  - Each **cell** can store **one symbol or** be "**blank**"
  - **Tape** is **initially** all **blank except** a few **cells** of the tape **containing the input string**
  - **Read/write head** can **scan one cell** of the tape - **starts on input**

- **In each step, a Turing machine**
  1. **Reads** the **currently scanned cell**
  2. **Based on current state and scanned symbol**
     i. **Overwrites symbol in scanned cell**
     ii. **Moves read/write head left or right one cell**
     iii. **Changes to a new state**

- **Each Turing Machine is specified by its finite set of rules**

Turing machines

	_	0	1
s_1	(1, L, s_3)	(1, L, s_4)	(0, R, s_2)
s_2	(0, R, s_1)	(1, R, s_1)	(0, R, s_1)
s_3			
s_4			



UW CSE's Steam-Powered Turing Machine

Original in Sieg Hall stairwell

Turing machines

Ideal Java/C programs:

- Just like the Java/C you're used to programming with, except you never run out of memory
  - no OutOfMemoryError

Equivalent to Turing machines but easier to program:

- Turing machine definition is useful for breaking computation down into simplest steps
- We only care about high level so we use programs

Turing's big idea part 1: Machines as data

Original Turing machine definition:

- A different "machine" **M** for each task
- Each machine **M** is defined by a finite set of possible operations on finite set of symbols
- So... **M** has a finite description as a sequence of symbols, its "code", which we denote **<M>**

You already are used to this idea with the notion of the program code, but this was a new idea in Turing's time.

Turing's big idea part 2: A Universal TM

- **A Turing machine interpreter U**
  - On input <M> and its input x,
    U outputs the same thing as M does on input x
  - At each step it decodes which operation M would have performed and simulates it.
- **One Turing machine is enough**
  - Basis for modern stored-program computer
    Von Neumann studied Turing's UTM design