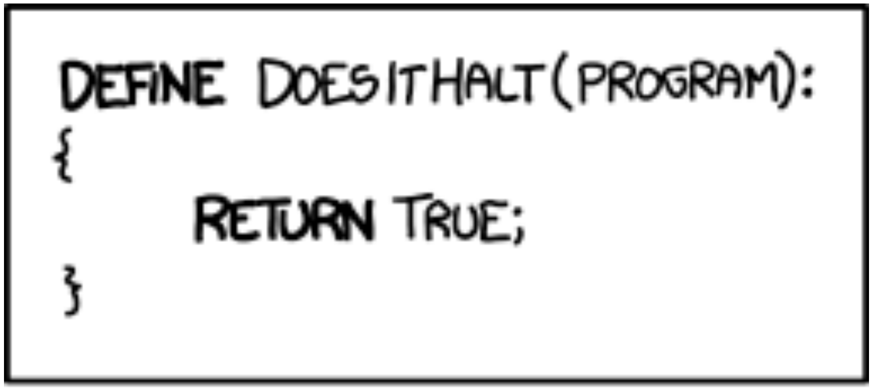
CSE 311: Foundations of Computing

Lecture 28: Undecidability



```
DEFINE DOESITHALT(PROGRAM):
{
      RETURN TRUE;
}
```

THE BIG PICTURE SOLUTION
TO THE HALTING PROBLEM

Final exam Monday, Review session Sunday

- **Monday** at either **2:30-4:20 (B)** or **4:30-6:20 (A)**
  - **CSE2 G20**
  - bring your **UW ID**
  - 1 hour and 50 minutes

- **Comprehensive:** Full probs only on topics that were covered in homework. May have small probs on other topics.
  - reference sheets will be included

- **Review session: Sunday at 3pm in CSE2 G20**
  - bring your questions

Final Exam

- 9 problems covering:
  - DFA / NFA / RE / CFG design
  - DFA / NFA / RE algorithms
  - Irregularity
  - Number theory
  - Set theory
  - Strong induction
  - Structural induction
  - Small questions on anything else
  - (any English proofs would be translations or templates)

Last time: Countable sets

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

$$\boldsymbol{S} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots\}$$

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”.

A note on this proof

- The set of rational numbers in [0,1) also have decimal representations like this
  - The only difference is that rational numbers always have repeating decimals in their expansions 0.33333… or .25000000…
- So why wouldn't the same proof show that this set of rational numbers is uncountable?
  - Given any listing we could create the flipped diagonal number $\boldsymbol{d}$ as before
  - However, $\boldsymbol{d}$ would not have a repeating decimal expansion and so wouldn't be a rational #

It would not be a "missing" number, so no contradiction.

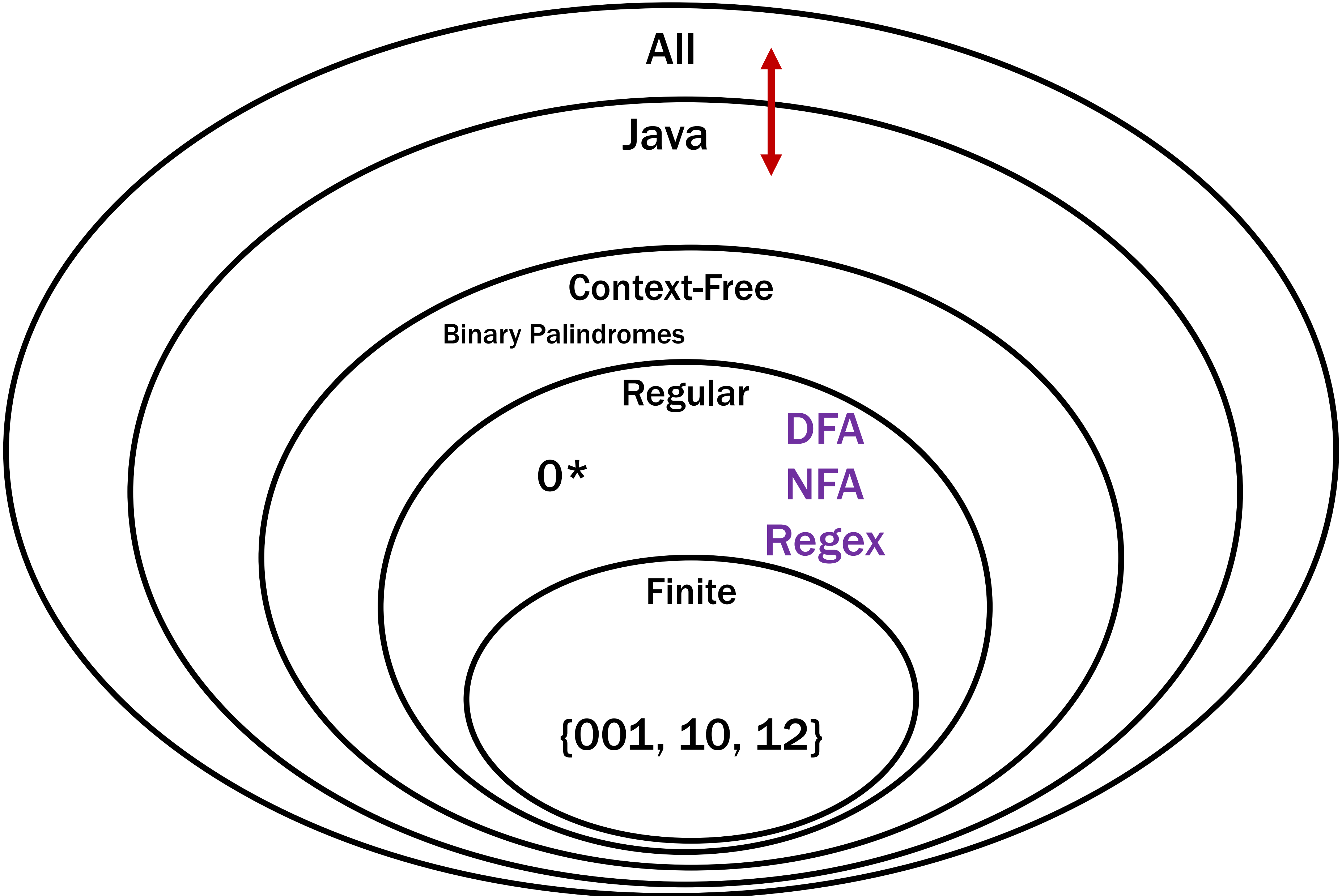
Uncomputable functions

We have seen that:

- The set of all (Java) programs is countable
- The set of all functions $f : \mathbb{N} \to \{0, \dots, 9\}$ is not countable

So: There must be some function $f : \mathbb{N} \to \{0, \dots, 9\}$ 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=10000000000000000001?

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**))?

“(((())))..;AACPSSaaabceeggghiiiilnnnnnooprrrrrrrrrrrsssttttttuuwxxyy{}”

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**

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**

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 s **D**(CODE(**D**)) **doesn’t halt**

Suppose th **doesn’t halt.**

Th of **H** it must be that

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

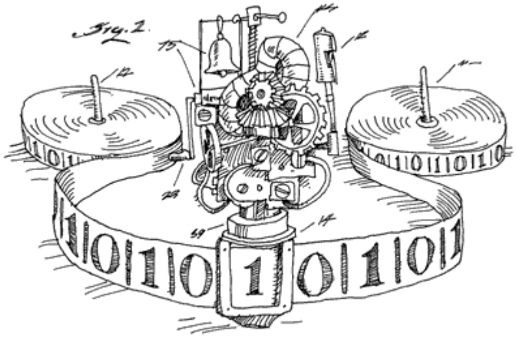
Whi by the definition of **D** means **D**(CODE(**D**)) **halts**

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.

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"

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**

All programs **P**

	<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							
.							
.							

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**

	$\langle P_1\rangle$	$\langle P_2\rangle$	$\langle P_3\rangle$	$\langle P_4\rangle$	$\langle P_5\rangle$	$\langle P_6\rangle$	….					
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**

All programs **P**

	$\langle P_1\rangle$	$\langle P_2\rangle$	$\langle P_3\rangle$	$\langle P_4\rangle$	$\langle P_5\rangle$	$\langle P_6\rangle$	….					
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	.	.	. .	.	.	.	.	.	.	.		.
.	.	.	. .	.	.	.	.	.	.	.		.
.												

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

(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 → Halting Problem decidable”

Contrapositive:

“Halting Problem undecidable → 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!