CSE 311: Foundations of Computing

Lecture 27: Undecidability

Final exam

- **Monday** at either **2:30-4:20 (B)** or **4:30-6:20 (A)**
  - **CSE2 G20**
  - Bring your **UW ID** and have it ready before the exam

- **Comprehensive** coverage. If you had a homework question on it, it is fair game. Small probs on other topics.
  - Includes pre-midterm topics, e.g. formal proofs.
    Will contain any useful reference sheets at end.

- **Review session: *Saturday* 1-3 pm in SAV 260**
  - **Bring your questions !!**

Final exam problems

8 problems covering the following:

- Formal proofs
- Induction: ordinary, strong, structural
- Language design: RE, DFA, NFA, CFG, recursive sets
- FSM algorithms: RE to NFA, NFA to DFA, state minimization
- Proving irregularity
- Small problems on other topics such as
  - Modular arithmetic
  - Relations
  - Set theory
  - 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" ($\mathbb{Z}$, $\mathbb{Q}$, Σ^*)

Last time: Not every set is countable

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

Proof using "diagonalization".

Last time: Proof that $[0,1)$ is not countable

Suppose, for the sake of contradiction, that there is a list of them:

		1	2	3	4						
r_1	0.	5^{1}	0	0	0						
r_2	0.	3	3^{5}	3	3						
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
						2^{5}	1	2	2	...	...
						0	0^{5}	0	0	...	...
						8	1	8^{5}	2	...	...

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

For every $n \geq 1$:

$r_n \neq d = 0.\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55}\cdots$

because the numbers differ on the n-th digit!

So the list is incomplete, which is a contradiction.

Thus the real numbers between 0 and 1 are **not countable**: "uncountable" ■

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 d as before
  - However, 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.

Last time:

The set of all functions $f : \mathbb{N} \to \{0, \dots, 9\}$ is uncountable

Supposed listing of all the functions:

	1	2	3	4						
f_1	5^{1}	0	0	0						
f_2	3	3^{5}	3	3						
f_3	1	4	2^{5}	8	5	7	1	4	...	...
f_4	1	4	1	5^{1}	9	2	6	5	...	...
f_5	1	2	1	2	2^{5}	1	2	2	...	...
f_6	2	5	0	0	0	0^{5}	0	0	...	...
f_7	7	1	8	2	8	1	8^{5}	2	...	...

Flipping rule:

If $f_n(n) = 5$, set $D(n) = 1$

If $f_n(n) \neq 5$, set $D(n) = 5$

For all n, we have $D(n) \neq f_n(n)$. Therefore $D \neq f_n$ for any n and the list is incomplete! $\Rightarrow$ $\{f \mid f: \mathbb{N} \to \{0,1,\dots,9\}\}$ is **not** countable

Last time: Uncomputable functions

We have seen that:

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

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

Interesting… maybe.

Can we come up with 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)
    }
}
```

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

What does this program do?

… on n=11?

… on n=10000000000000000001?

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)
    }
}
```

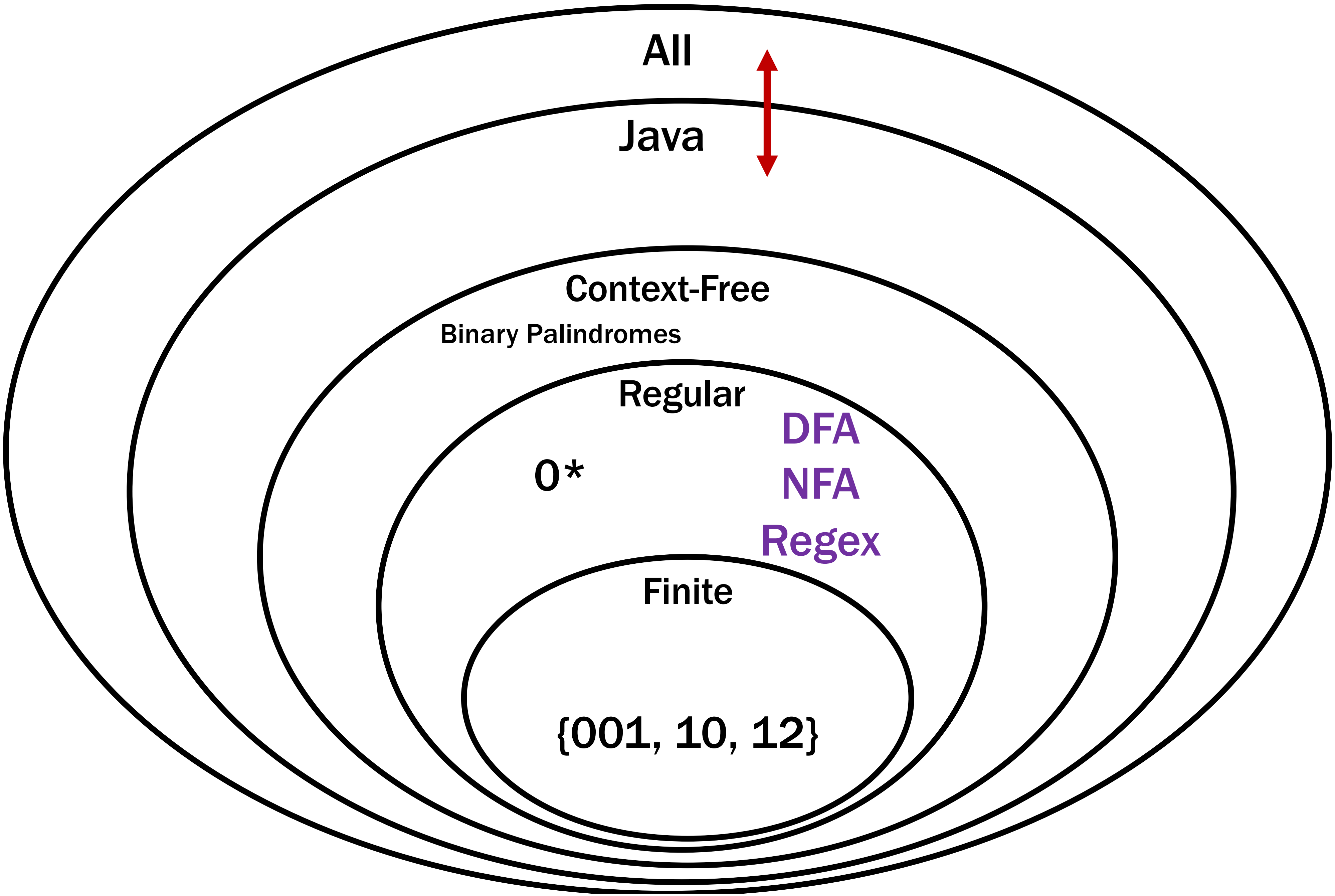
Nobody knows whether or not this program halts on all inputs!

What does this program do?

… on `n=11?`

… on `n=100000000000000000001?`

Recall our language picture



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**))?

"(((()))))..;AACPSSaaabceeggghiiiilnnnnnooprrrrrrrrrrsssttttttuuwxxyy{}"

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) == true) {
        ...
    } else {
        ...
    }
}

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) == true) {
        ...
    } else {
        ...
    }
}
```

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

```
public static void D(s) {
    if (H(s,s) == true) {
        ...
    } else {
        ...
    }
}
```

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) == true) {
        while (true);  // don't halt
    } else {
        ...
    }
}
```

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) == true) {
        while (true);  // don't halt
    } else {
        ...
    }
}
```

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) == true) {
        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) == true) {
        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) == true) {
        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 mu…

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**

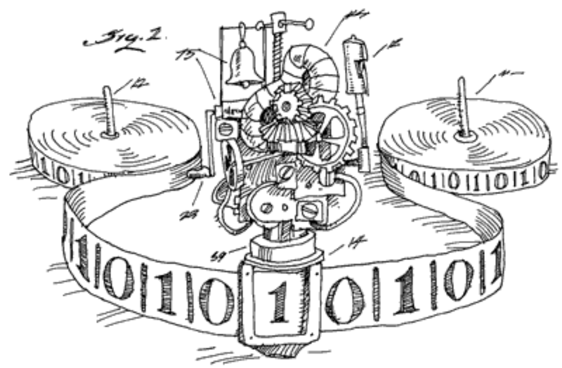
Whi… by the definition of D means D(CODE(D)) **halts**

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**

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**

	$<P_1>$	$<P_2>$	$<P_3>$	$<P_4>$	$<P_5>$	$<P_6>$	....					
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**

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	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.												

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:

Prove that if there were a program deciding B then there would be a way to build 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 141 assignment

Students should write a Java program that:

- Prints “Hello” to the console
- Eventually exits

GradeIt, PracticeIt, etc. need to grade the students.

How do we write that grading program?

WE CAN'T: THIS IS IMPOSSIBLE!

A related undecidable problem

- **HelloWorldTesting Problem:**
  - **Input: CODE(Q)** and **x**
  - **Output:**

    True if **Q** outputs "HELLO WORLD" on input **x**

    False if **Q** does not output "HELLO WORLD" on input **x**

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

A related undecidable problem

- Suppose there is a program T that solves the HelloWorldTesting problem. Define program H that takes input CODE(P) and x and does the following:
  - Creates CODE(Q) from CODE(P) by
    - (1) removing all output statements from CODE(P), and
    - (2) adding a `System.out.println("HELLO WORLD")` immediately before any spot where P could halt

    Then runs T on input CODE(Q) and x.

A related undecidable problem

```
public class Q {
  public static void main(String[] args) {
    PrintStream out = System.out;
    System.setOut(new PrintStream(
        new WriterOutputStream(new StringWriter()));

    P(args);

    out.println(“HELLO WORLD”);
  }
}

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

A related undecidable problem

- **Suppose there is a program T that solves the HelloWorldTesting problem. Define program H that takes input CODE(P) and x and does the following:**
  - **Creates CODE(Q) from CODE(P) by**
    - **(1) removing all output statements from CODE(P), and**
    - **(2) adding a** System.out.println("HELLO WORLD") **immediately before any spot where P could halt**

    Then runs T on input CODE(Q) and x.

- If **P halts on input x** then **Q** prints HELLO WORLD and halts and so H outputs **true** (because T outputs true on input **CODE(Q)**)
- If **P doesn't halt on input x** then **Q** won't print anything since we removed any other print statement from CODE(**Q**) so H outputs **false**

We know that such an H cannot exist. Therefore T cannot exist.

The HaltsNoInput Problem

- **Input: CODE(R)** for program **R**
- **Output: True** if **R** halts without reading input

  False otherwise.

Theorem: HaltsNoInput is undecidable

General idea "hard-coding the input":

- Show how to use **CODE(P)** and **x** to build **CODE(R)** so **P halts on input x ⇔ R halts without reading input**

The HaltsNoInput

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

  public static void main(String[] args) {
    System.setIn(new ReaderInputStream(
        new StringReader(x)));

    P(args);
  }
}

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

The HaltsNoInput Problem

“Hard-coding the input”:

- Show how to use **CODE(P)** and **x** to build **CODE(R)** so **P halts on input x ⇔ R halts without reading input**

- So if we have a program **N** to decide **HaltsNoInput** then we can use it as a subroutine as follows to decide the Halting Problem, which we know is impossible:
  - On input CODE(**P**) and **x**, produce CODE(**R**). Then run **N** on input CODE(**Q**) and output the answer that **N** gives.

CSE 141 grading is impossible

- The impossibility of writing the CSE 141 grading program follows by combining the ideas from the undecidability of HaltsNoInput and HelloWorld.

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 (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.

But many problems about CFGs are undecidable!

- **Is there any string that two CFGs both accept?**
- Do two CFGs accept the same language?
- **Does a CFG accept *every* string?**

More Complexity Theory

Not just what can be computed at all...

How about what can be computed *efficiently*?

A rich, interesting, and important topic.

See CSE 431 for much more on that!

Final exam

- **Monday** at either **2:30-4:20 (B)** or **4:30-6:20 (A)**
  - **CSE2 G20**
  - Bring your **UW ID** and have it ready before the exam
- **Comprehensive** coverage. If you had a homework question on it, it is fair game. (Maybe small probs on other topics.)
  - Includes pre-midterm topics, e.g. formal proofs.
    Will contain any useful reference sheets at end.
- **Review session: *Saturday* 1-3 pm in SAV 260**
  - **Bring your questions !!**