CSE 311: Foundations of Computing

Lecture 28: Undecidability

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**

- **Comprehensive:** Full probs only on topics that were covered in homework. May have small probs on other topics.
  - May includes pre-midterm topics, e.g., formal proofs.
  - Reference sheets will be included.

- **Review session:** ***Sunday*** **1-3 pm in CSE2 G20**
  - **Bring your questions !!**

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”.

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

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

		1	2	3	4	5	6	7	8	9	...
r_1	0.	5	0	0	0	0	0	0	0	...	...
r_2	0.	3	3	3	3	3	3	3	3	...	...
r_3	0.	1	4	2	8	5	7	1	4	...	...
r_4	0.	1	4	1	5	9	2	6	5	...	...
r_5	0.	1	2	1	2	2	1	2	2	...	...
r_6	0.	2	5	0	0	0	0	0	0	...	...
r_7	0.	7	1	8	2	8	1	8	2	...	...
r_8	0.	6	1	8	0	3	3	9	4	...	...
...	....	...	....	....	...	...	...	...	...	...	

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

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

		1	2	3	4	5	6	7	8	9	...
r_1	0.	5	0	0	0	0	0	0	0	...	...
r_2	0.	3	3	3	3	3	3	3	3	...	...
r_3	0.	1	4	2	8	5	7	1	4	...	...
r_4	0.	1	4	1	5	9	2	6	5	...	...
r_5	0.	1	2	1	2	2	1	2	2	...	...
r_6	0.	2	5	0	0	0	0	0	0	...	...
r_7	0.	7	1	8	2	8	1	8	2	...	...
r_8	0.	6	1	8	0	3	3	9	4	...	...
...	....	...	....	....	...	...	...	...	...	...	

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

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

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

		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	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...
r_8	0.	6	1	8	0	3	3	9	4^{5}	...	...
...	....	...	....	....	...	...	...	...	...	...	

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

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

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

		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	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...

If diagonal element is $\mathbf{0.}\,x_{11}x_{22}x_{33}x_{44}x_{55}\cdots$ then let's call the flipped number $\mathbf{0.}\,\hat{x}_{11}\hat{x}_{22}\hat{x}_{33}\hat{x}_{44}\hat{x}_{55}\cdots$

It cannot appear anywhere on the list!

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

Suppose, for a 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	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...

Flipping rule:

If digit is 5, make it 1.

If digit is not 5, make it 5.

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.

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

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

Supposed listing of all the functions:

	1	2	3	4	5	6	7	8	9	…
f_1	5	0	0	0	0	0	0	0	…	…
f_2	3	3	3	3	3	3	3	3	…	…
f_3	1	4	2	8	5	7	1	4	…	…
f_4	1	4	1	5	9	2	6	5	…	…
f_5	1	2	1	2	2	1	2	2	…	…
f_6	2	5	0	0	0	0	0	0	…	…
f_7	7	1	8	2	8	1	8	2	…	…
f_8	6	1	8	0	3	3	9	4	…	…
…	…	….	….	…	…	…	…	…	…	

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	...	...
f_8	6	1	8	0	3	3	9	4^{5}	...	...
...	...	....	....	...	...	...	...	...	...	

Flipping rule:

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

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

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

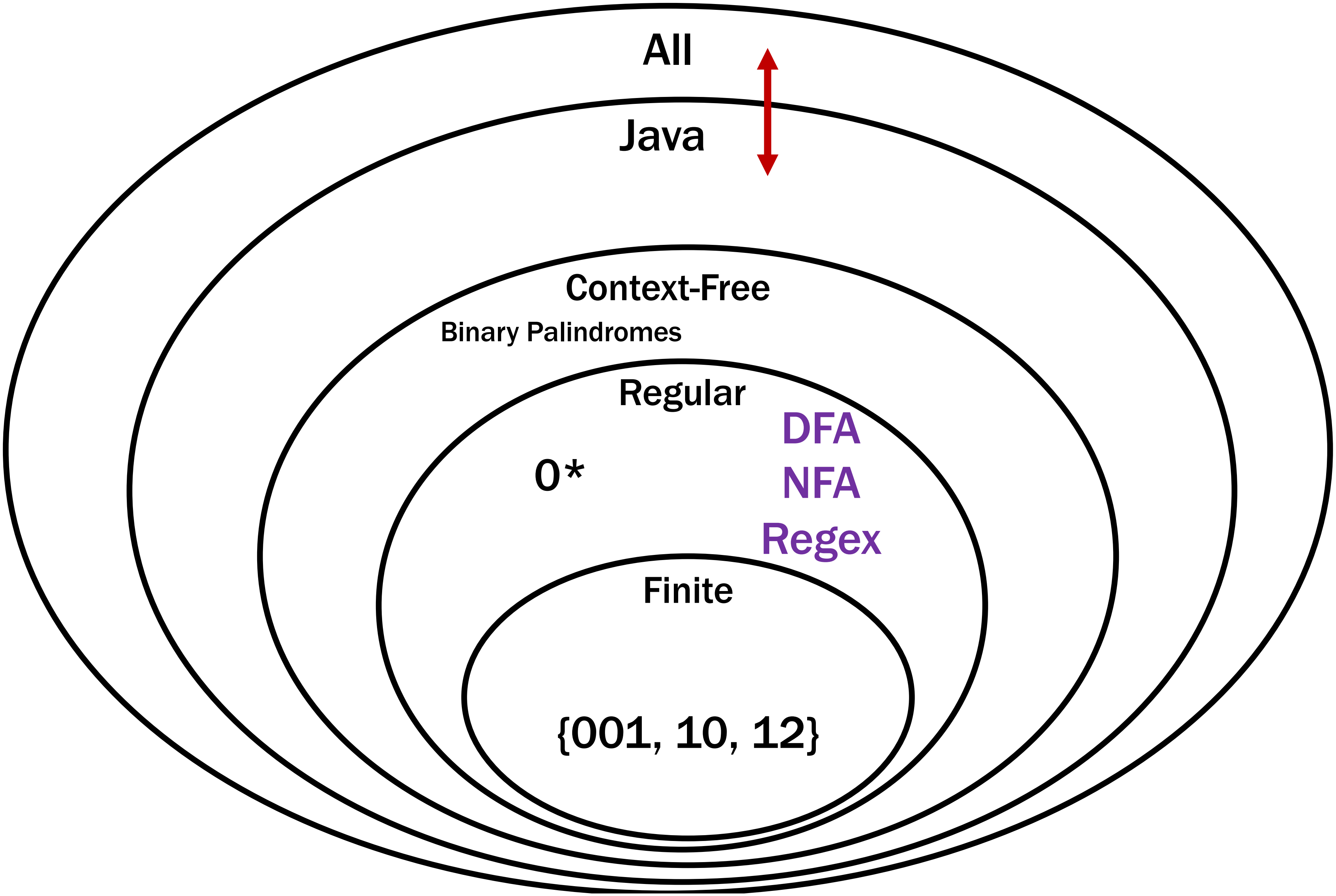
Uncomputable functions

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!

Recall our language picture



Uncomputable functions

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=1000000000000000001?

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) == 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

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

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) == 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**

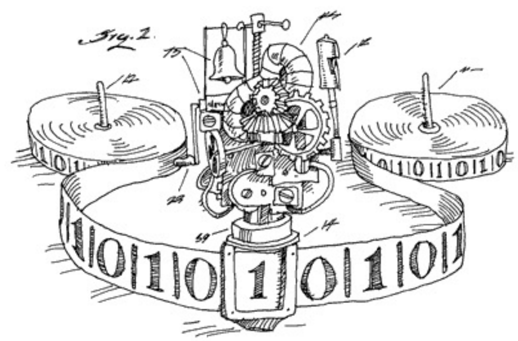
Whi... by the definition of D means D(CODE(D)) **halts**

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.

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**

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

All programs **P**

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)