

CSE 311: Foundations of Computing

Fall 2014

Lecture 26: Pattern matching, Limits of Computation

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

THE BIG PICTURE SOLUTION
TO THE HALTING PROBLEM

Pattern Matching

- **Given**
 - A string, **s**, of **n** characters
 - A pattern, **p**, of **m** characters
 - usually $m \ll n$
- **Find**
 - All occurrences of the pattern **p** in the string **s**
- **Obvious algorithm:**
 - Check if **p** matches at each of the positions in **s**
Stop at a failed match and try the next position

string **s** = x y x x y x y x y y x y x y x y y x y x y x x

pattern **p** = x y x y y x y x y x x

string **s** = **x y x x** y x y x y y x y x y x y y x y x y x x
x y x y y x y x y x x

string **s** = x y x x y x y x y y x y x y x y y x y x y x x
x y x y
x y x y y x y x y x x

string **s** = x y **x** **x** y x y x y y x y x y x y y x y x y x x
x y x y
x
x y x y y x y x y x x

string **s** = x y x x y x y x y y x y x y x y y x y x y x x
x y x y
x
x y
x y x y y x y x y x x

string **s** = x y x x **y** x y x y y x y x y x y y x y x y x x
x y x y
x
x y
x y x y y
x y x y y x y x y x x

string **s** = x y x x y x y x y y x y x y x y y x y x y x x
x y x y
x
x y
x y x y y
x
x y x y y x y x y x x

string **s** = x y x x y x **y** x y y x y x y x y y x y x y x x
x y x y
x
x y
x y x y y
x
x y x y y x y x y x x
x y x y y x y x y x x

string **s** = x y x x y x y x y x y x y x y y x y x y x x
x y x y
x
x y
x y x y y
x
x y x y y x y x y x x
x
x y x y y x y x y x x

string **s** = x y x x y x y x **y** y x y x y x y y x y x y x x

x y x y

x

x y

x y x y y

x

x y x y y x y x y x x

x

x y x

x y x y y x y x y x x

string **s** = x y x x y x y x y **y** x y x y x y y x y x y x x

x y x y

x

x y

x y x y y

x

x y x y y x y x y x x

x

x y x

x

x y x y y x y x y x x

string **s** = x y x x y x y x y y x y x y x y y x y x y x x

x y x y

x

x y

x y x y y

x

x y x y y x y x y x x

x

x y x

x

x

x y x y y x y x y x x

string **s** = x y x x y x y x y y x **y** x y x y y x y x y x x

x y x y

x

x y

x y x y y

x

x y x y y x y x y x x

x

x y x

x

x

x y x y y

x y x y y x y x y x x

string **s** = x y x x y x y x y y x y x y x y y x y x y x x

x y x y

x

x y

x y x y y

x

x y x y y x y x y x x

x

x y x

x

x

x y x y y

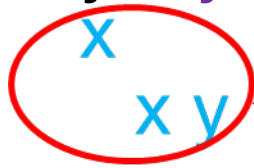
x

x y x y y x y x y x x

Worst-case time
 $O(mn)$

string **s** = x y x x y x y x y y x y x y x y y x y x y x x

x y x y

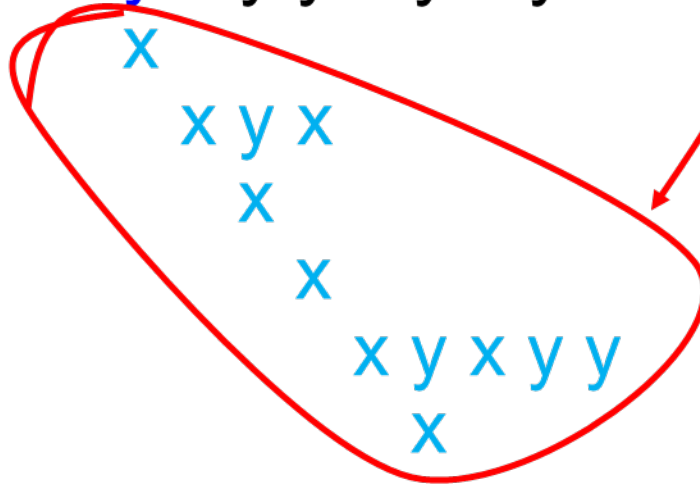


Lots of wasted work

x y x y y



x y x y y x y x y x x



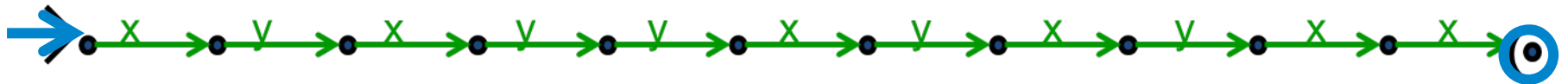
x y x y y x y x y x x

Better Pattern Matching via Finite Automata

- **Build a DFA for the pattern (preprocessing) of size $O(m)$**
 - Keep track of the ‘longest match currently active’
 - The DFA will have only $m+1$ states
- **Run the DFA on the string n steps**
- **Obvious construction method for DFA will be $O(m^2)$ but can be done in $O(m)$ time.**
- **Total $O(m+n)$ time**

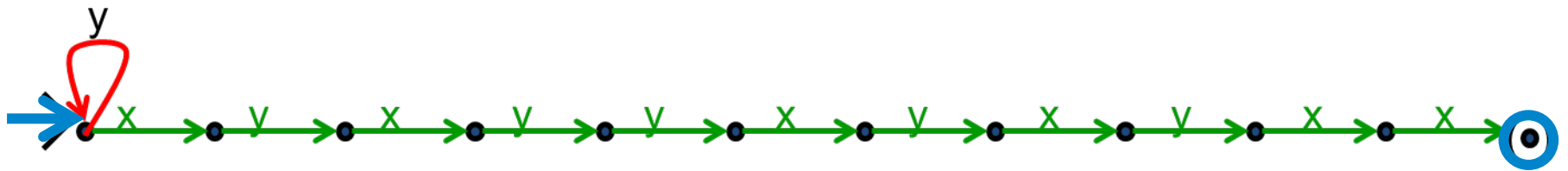
Building a DFA for the Pattern

pattern **p**=x y x y y x y x y x x



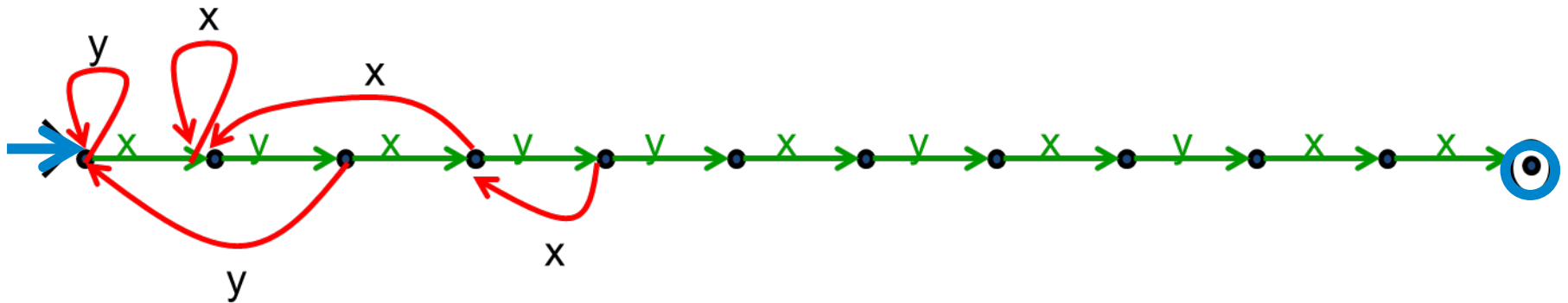
Preprocessing The Pattern

pattern $\mathbf{p} = x y x y y x y x y x x$



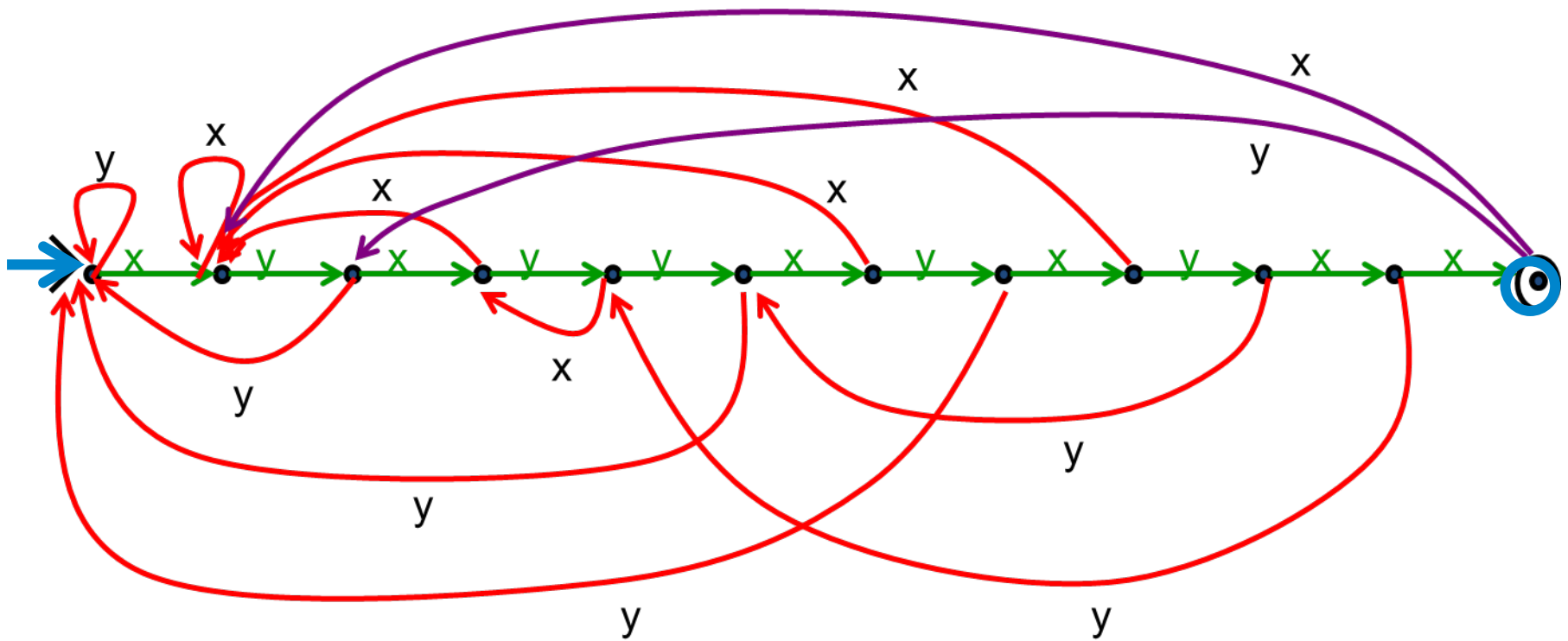
preprocessing the pattern

pattern $\mathbf{p} = x y x y y x y x y x x$



preprocessing the pattern

pattern $\mathbf{p} = x y x y y x y x y x x$

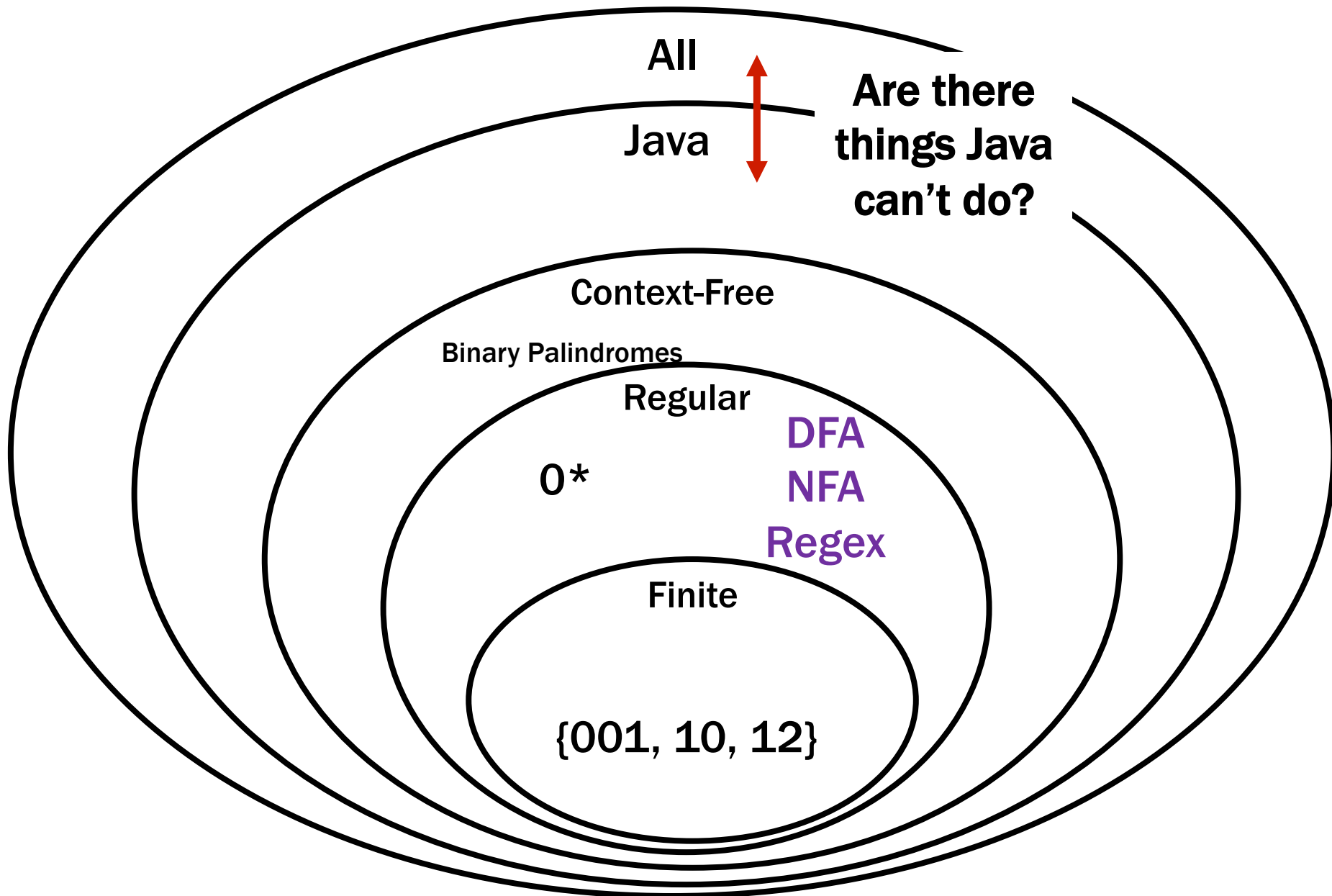


Generalizing

- **Can search for arbitrary combinations of patterns**
 - Not just a single pattern
 - **Build NFA for pattern then convert to DFA ‘on the fly’.**

Compare DFA constructed above with subset construction for the obvious NFA.

Languages and Machines!



An Assignment Too Simple for 142!

Students should write a Java program that...

- Prints “Hello” to the console**
- Eventually exits**

Gradel, Practicel, etc. need to grade the students.

How do we write that grading program?

Follow Up Question

What does this program do?

```
_( __, __, __ ) { __ / __ <= 1 ? _ ( __, __
+1, __ ) : ! ( __ % __ ) ? _ ( __, __
+1, 0 ) : __ % __ == __ / __ && ! __ ?
( printf ( "%d\t", __ / __ ), _ ( __, __
+1, 0 ) ) : __ % __ > 1 && __ % __ < __ / __ ?
_ ( __, 1 + __, __ + ! ( __ / __ % ( __
% __ ) ) ) : __ < __ * __ ? _ ( __, __ + 1, __ ) :
0 ; } main () { _ ( 100, 0, 0 ) ; }
```


Sneak Peak

It turns out the simple autograder is impossible to write...

And we'll prove it!

Some Notation and Starting Ideas

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

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**))?

“((()))..;AACPS SaaabceeggghiiiiInnnnnnoopr rrrrrrrrrrrrssstttttuuwxyy{ }”

The Halting Problem

Given:

- $\text{CODE}(\mathbf{P})$ for a program \mathbf{P}
- Input \mathbf{x}

Output:

- **true** if \mathbf{P} halts on input \mathbf{x}
- **false** if \mathbf{P} does not halt on input \mathbf{x}

The Halting Problem

Given:

- $\text{CODE}(\mathbf{P})$ for a program \mathbf{P}
- Input \mathbf{x}

Output:

- **true** if \mathbf{P} halts on input \mathbf{x}
- **false** if \mathbf{P} does not halt on input \mathbf{x}

It turns out that it isn't possible to write a 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(x) {  
    if (H(x,x) == true) {  
        while (true);    /* don't halt */  
    }  
    else {  
        return;          /* halt */  
    }  
}
```

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

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

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

H solves the halting problem implies that

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

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

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

H solves the halting problem implies that

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

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

Then, we must be in the **second** case of the if.

So, **H**(CODE(**D**), CODE(**D**)) is **false**

Which means **D**(CODE(**D**)) **doesn't halt**

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

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

H solves the halting problem implies that

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

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

Then, we must be in the **second** case of the if.

So, **H**(CODE(**D**), CODE(**D**)) is **false**

Which means **D**(CODE(**D**)) **doesn't halt**

Suppose **D**(CODE(**D**)) **doesn't halt**.

Then, we must be in the **first** case of the if.

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

Which means **D**(CODE(**D**)) **halts**.

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

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

H solves the halting problem implies that

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

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

Then, we must be in the **second** case of the if.

So, **H**(CODE(**D**), CODE(**D**)) is **false**

Which means **D**(CODE(**D**)) **doesn't halt**

Suppose **D**(CODE(**D**)) **doesn't halt**.

Then, we must be in the **first** case of the if.

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

Which means **D**(CODE(**D**)) **halts**.



That's it!

- **We proved that there is no computer program that can solve the Halting Problem.**
 - **There was nothing special about Java**
- **This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have.**

What's next?

- **We showed: If some “hypothetical” subroutine H existed that solved the Halting Problem then it would let us build a program D that cannot possibly exist**
 - We will use the same idea to show that programs solving other problems are impossible, but we now will be able to use that H cannot exist
- **A key piece of the proof was considering what a program does when given its own code as input**
 - This was inspired by a method to compare the sizes of infinite sets call *diagonalization* that we will study next class