Lecture 11: Self-Reference and Uncomputability

Self-Referential Subtitles Are Best Subtitles

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Ancient Cretan says "All Cretans are liars"

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Ancient Cretan says "All Cretans are liars"

- Are they lying?

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Ancient Cretan says "All Cretans are liars"

- Are they lying?

Barber says "I shave those who don't themselves"

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Ancient Cretan says "All Cretans are liars"

- Are they lying?

Barber says "I shave those who don't themselves"

- Does the barber shave themself?

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Ancient Cretan says "All Cretans are liars"

- Are they lying?

Barber says "I shave those who don't themselves"

- Does the barber shave themself?

I say "This statement is false"

Liar's Paradox

Self-Reference: "the act or an instance of referring or alluding to oneself; see self-reference"

Can create issues in logical deduction!

Ancient Cretan says "All Cretans are liars"

- Are they lying?

Barber says "I shave those who don't themselves"

- Does the barber shave themself?

I say "This statement is false"

- Is it?

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Yes?

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Yes?

- If $S \in S$, S defined to *not* include S!

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Yes?

- If $S \in S$, S defined to *not* include S!

No?

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Yes?

- If $S \in S$, S defined to *not* include S!

No?

- If $S \notin S$, S defined to include S!

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Yes?

- If $S \in S$, S defined to *not* include S!

No?

- If $S \notin S$, S defined to include S!

Set theory solution: make sure S not definable

Russell's Paradox

Let S be set of sets that don't contain themselves
$S = \{x \mid x \notin x\}$

Does S contain itself? Is $S \in S$?

Yes?

- If $S \in S$, S defined to *not* include S!

No?

- If $S \notin S$, S defined to include S!

Set theory solution: make sure S not definable

In CS, not so easy to avoid!

An Important Aside

Computer programs $\equiv$ binary strings

An Important Aside

Computer programs $\equiv$ binary strings

```python
import sys

def example_function():
    print("This is an example function")
    print("It does example things")
    sys.crash()
```

```
696d 706f 7274 2073 7973 0d0a 0d0a 6465
6620 6578 616d 706c 655f 6675 6e63 7469
6f6e 2829 3a0d 0a09 7072 696e 7428 2254
6869 7320 6973 2061 6e20 6578 616d 706c
6520 6675 6e63 7469 6f6e 2229 0d0a 0970
7269 6e74 2822 4974 2064 6f65 7320 6578
616d 706c 6520 7468 696e 6773 2229 0d0a
0973 7973 2e63 7261 7368 2829 0d0a 0d0a
0d0a 09
```

An Important Aside

Computer programs $\equiv$ binary strings

```python
import sys

def example_function():
    print("This is an example function")
    print("It does example things")
    sys.crash()
```

```
696d 706f 7274 2073 7973 0d0a 0d0a 6465
6620 6578 616d 706c 655f 6675 6e63 7469
6f6e 2829 3a0d 0a09 7072 696e 7428 2254
6869 7320 6973 2061 6e20 6578 616d 706c
6520 6675 6e63 7469 6f6e 2229 0d0a 0970
7269 6e74 2822 4974 2064 6f65 7320 6578
616d 706c 6520 7468 696e 6773 2229 0d0a
0973 7973 2e63 7261 7368 2829 0d0a 0d0a
0d0a 09
```

Means we can pass programs as inputs to programs

An Important Aside

Computer programs $\equiv$ binary strings

```python
import sys

def example_function():
    print("This is an example function")
    print("It does example things")
    sys.crash()
```


```
696d 706f 7274 2073 7973 0d0a 0d0a 6465
6620 6578 616d 706c 655f 6675 6e63 7469
6f6e 2829 3a0d 0a09 7072 696e 7428 2254
6869 7320 6973 2061 6e20 6578 616d 706c
6520 6675 6e63 7469 6f6e 2229 0d0a 0970
7269 6e74 2822 4974 2064 6f65 7320 6578
616d 706c 6520 7468 696e 6773 2229 0d0a
0973 7973 2e63 7261 7368 2829 0d0a 0d0a
0d0a 09
```


Means we can pass programs as inputs to programs
Program can be own input — allows self-reference!

An Impossible Problem

Halting problem: determine if program halts

Formally, want program `TestHalt` such that

- If `P(x)` halts, `TestHalt(P, x)` = True
- If `P(x)` loops, `TestHalt(P, x)` = False

An Impossible Problem

Halting problem: determine if program halts

Formally, want program `TestHalt` such that

- If `P(x)` halts, `TestHalt(P, x)` = True
- If `P(x)` loops, `TestHalt(P, x)` = False

Thm: Problem undecidable – `TestHalt` can't exist!

An Impossible Problem

Halting problem: determine if program halts

Formally, want program `TestHalt` such that

- If `P(x)` halts, `TestHalt(P, x)` = True
- If `P(x)` loops, `TestHalt(P, x)` = False

Thm: Problem undecidable – `TestHalt` can't exist!

To prove: assume for contradiction `TestHalt` exists
Use self-reference to defeat `TestHalt`

Turing The Computer Scientist

Assume for contradiction `TestHalt` exists

```
Turing(P):
   if TestHalt(P, P) = True:
       loop infinitely
   else:
       halt
```

Turing The Computer Scientist

Assume for contradiction `TestHalt` exists

```
Turing(P):
    if TestHalt(P, P) = True:
        loop infinitely
    else:
        halt
```

What does `Turing(Turing)` do?

Turing The Computer Scientist

Assume for contradiction `TestHalt` exists

```
Turing(P):
    if TestHalt(P, P) = True:
        loop infinitely
    else:
        halt
```

What does `Turing(Turing)` do?

Opposite of `TestHalt(Turing, Turing)`

Turing The Computer Scientist

Assume for contradiction `TestHalt` exists

```
Turing(P):
    if TestHalt(P, P) = True:
        loop infinitely
    else:
        halt
```

What does `Turing(Turing)` do?

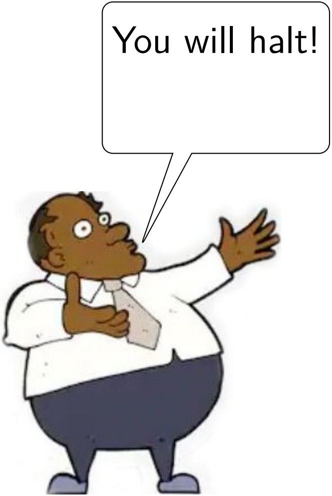
Opposite of `TestHalt(Turing, Turing)`
So `TestHalt` must be wrong there!

Turing The Sassy Teenager

`TestHalt`

`Turing`

Turing The Sassy Teenager



`TestHalt`

`Turing`

Turing The Sassy Teenager

You will halt!

I’ll loop instead!

`TestHalt`

`Turing`

Turing The Sassy Teenager

You will halt!

Fine, loop!

I'll loop instead!

`TestHalt`

`Turing`

Turing The Sassy Teenager

You will halt!

Fine, loop!

I'll loop instead!

Actually, I'll halt!

`TestHalt`

`Turing`

But Wait!

Why can't we just simulate `P(x)` and wait for halt?

But Wait!

Why can't we just simulate `P(x)` and wait for halt?

Might have to wait forever
But `TestHalt` must return in finite time!

But Wait!

Why can't we just simulate `P(x)` and wait for halt?

Might have to wait forever
But `TestHalt` must return in finite time!

What if I just wait 9000 years?

But Wait!

Why can't we just simulate `P(x)` and wait for halt?

Might have to wait forever
But `TestHalt` must return in finite time!

What if I just wait 9000 years?
`P(x)` might need 9001!

OK Sure, But...

...maybe `TestHalt` is just contrived?
Don't often care what program does on itself

OK Sure, But...

...maybe `TestHalt` is just contrived?
Don't often care what program does on itself

Perhaps better: does program halt with no input?

OK Sure, But...

...maybe `TestHalt` is just contrived?
Don't often care what program does on itself

Perhaps better: does program halt with no input?

"Easy" Halting Problem: want `ETH` such that

- If `P()` halts, `ETH(P)` = True
- If `P()` loops, `ETH(P)` = False

OK Sure, But...

...maybe `TestHalt` is just contrived?
Don't often care what program does on itself

Perhaps better: does program halt with no input?

"Easy" Halting Problem: want `ETH` such that

- If `P()` halts, `ETH(P)` = True
- If `P()` loops, `ETH(P)` = False

Claim: "Easy" Halting Problem no easier!

OK Sure, But...

...maybe `TestHalt` is just contrived?
Don't often care what program does on itself

Perhaps better: does program halt with no input?

"Easy" Halting Problem: want `ETH` such that

- If `P()` halts, `ETH(P)` = True
- If `P()` loops, `ETH(P)` = False

Claim: "Easy" Halting Problem no easier!

Formally: if `ETH` exists, `TestHalt` does too

Easy My ***

Suppose `ETH` exists, can write `TestHalt`:

```
TestHalt(P, x):
    def P'():
        P(x)
    return ETH(P')
```

Easy My ***

Suppose `ETH` exists, can write `TestHalt`:

```
TestHalt(P, x):
    def P'():
        P(x)
    return ETH(P')
```

Input or none doesn't matter — can just hardcode!

Easy My ***

Suppose `ETH` exists, can write `TestHalt`:

```
TestHalt(P, x):
    def P'():
        P(x)
    return ETH(P')
```

Input or none doesn't matter — can just hardcode!

In logic: `ETH` exists $\Longrightarrow$ `TestHalt` exists

Easy My ***

Suppose `ETH` exists, can write `TestHalt`:

```
TestHalt(P, x):
    def P'():
        P(x)
    return ETH(P')
```

Input or none doesn't matter — can just hardcode!

In logic: `ETH` exists $\Longrightarrow$ `TestHalt` exists
Contrapos: `TestHalt` doesn't exist $\Longrightarrow$ `ETH` doesn't

Easy My ***

Suppose `ETH` exists, can write `TestHalt`:

```
TestHalt(P, x):
    def P'():
        P(x)
    return ETH(P')
```

Input or none doesn't matter — can just hardcode!

In logic: `ETH` exists $\implies$ `TestHalt` exists
Contrapos: `TestHalt` doesn't exist $\implies$ `ETH` doesn't

Already Know `TestHalt` doesn't exist!

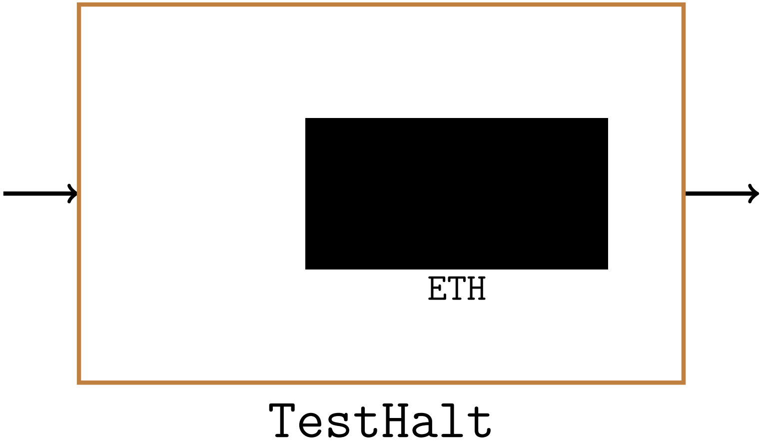
Reductionism

What we just did called a *reduction*

`TestHalt`

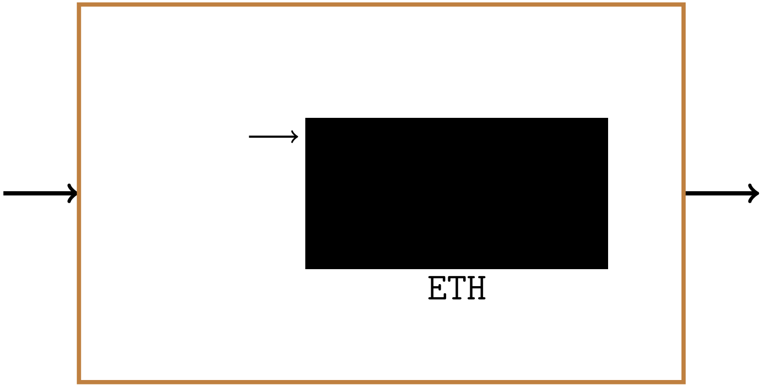
Reductionism

What we just did called a *reduction*



Reductionism

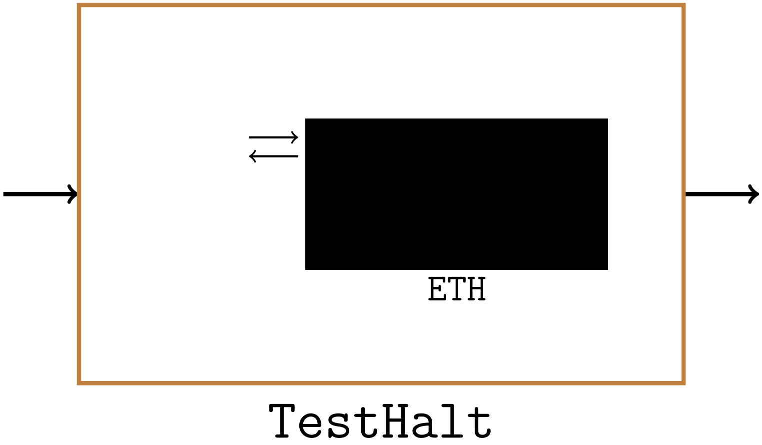
What we just did called a *reduction*



`TestHalt`

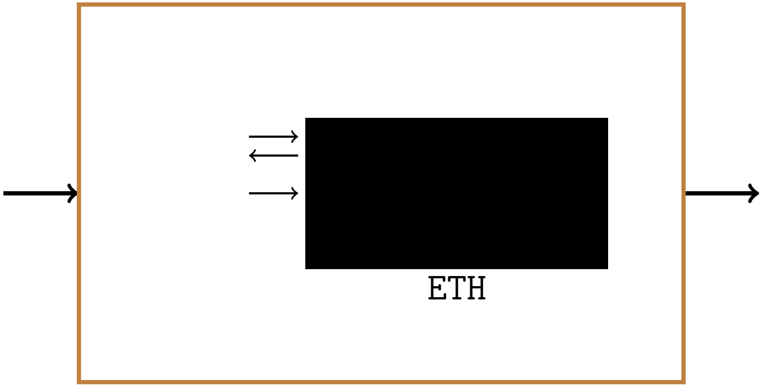
Reductionism

What we just did called a *reduction*



Reductionism

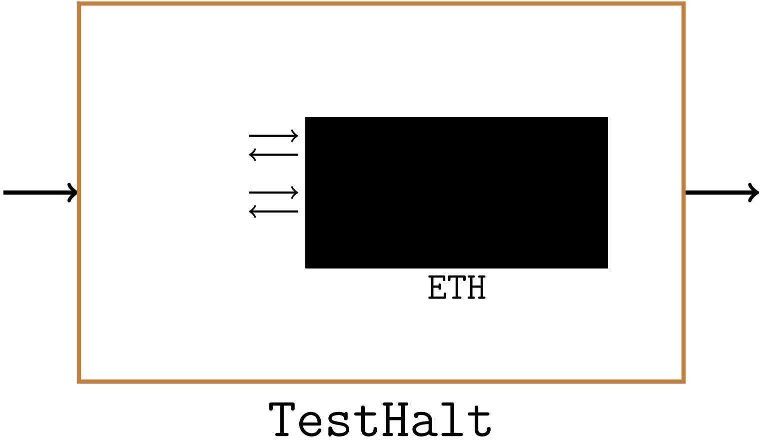
What we just did called a *reduction*



TestHalt

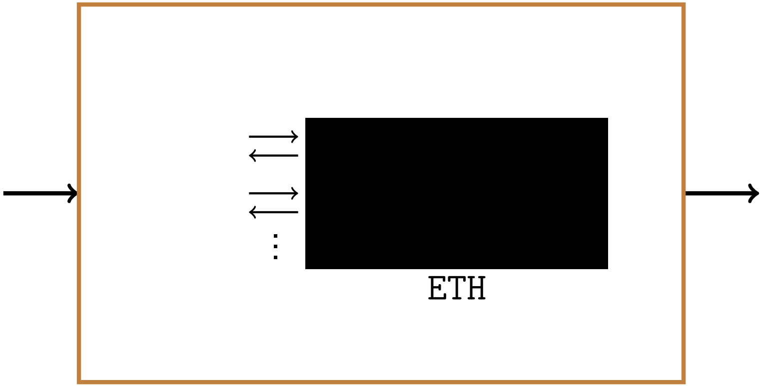
Reductionism

What we just did called a *reduction*



Reductionism

What we just did called a *reduction*



TestHalt

Reduce To The Problem Of Break Time

Time for a 4-minute break!

Reduce To The Problem Of Break Time

Time for a 4-minute break!

Today's Discussion Question:
If you were to write a self-referential discussion question, what would it be?

Recursive Enumerability

What happens if we relax the requirements?

[1]Sometimes called *recognizable*, but that doesn't sound as cool.

Recursive Enumerability

What happens if we relax the requirements?

Problem is *recursively enumerable*[1] if $\exists$ program `P`

- If answer for x is true, `P(x)` outputs true
- If answer is false, `P(x)` outputs false *or loops*

[1] Sometimes called *recognizable*, but that doesn't sound as cool.

Recursive Enumerability

What happens if we relax the requirements?

Problem is *recursively enumerable*[1] if $\exists$ program `P`

- If answer for x is true, `P(x)` outputs true
- If answer is false, `P(x)` outputs false *or loops*

Previously showed that Halting Prob is RE!

[1] Sometimes called *recognizable*, but that doesn't sound as cool.

Recursive Enumerability

What happens if we relax the requirements?

Problem is *recursively enumerable*[1] if $\exists$ program `P`

- If answer for x is true, `P(x)` outputs true
- If answer is false, `P(x)` outputs false *or loops*

Previously showed that Halting Prob is RE!

Can we find others?

[1] Sometimes called *recognizable*, but that doesn't sound as cool.

Entscheidungsproblem

Hilbert's famous "decision problem" (roughly):
Given a statement x, is it true or false?

Entscheidungsproblem

Hilbert's famous "decision problem" (roughly):
Given a statement x, is it true or false?

Claim: Entscheidungsproblem is undecidable

Entscheidungsproblem

Hilbert's famous "decision problem" (roughly):
Given a statement x, is it true or false?

Claim: Entscheidungsproblem is undecidable

Proof:
Suppose $\exists$ program `E` solving Entscheidungsproblem

Entscheidungsproblem

Hilbert's famous "decision problem" (roughly):
Given a statement x, is it true or false?

Claim: Entscheidungsproblem is undecidable

Proof:
Suppose $\exists$ program `E` solving Entscheidungsproblem

```
TestHalt(P, x):
    return E("P(x) halts.")
```

Entscheidungsproblem

Hilbert's famous "decision problem" (roughly):
Given a statement x, is it true or false?

Claim: Entscheidungsproblem is undecidable

Proof:
Suppose $\exists$ program `E` solving Entscheidungsproblem

```
TestHalt(P, x):
    return E("P(x) halts.")
```

Allows us to solve Halting Problem — no bueno!

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Proof:

- Try all proofs with one step

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Proof:

- Try all proofs with one step
- If none succeed, try all with two steps

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Proof:

- Try all proofs with one step
- If none succeed, try all with two steps
- Next try all with three steps

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Proof:

- Try all proofs with one step
- If none succeed, try all with two steps
- Next try all with three steps
- ...

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Proof:

- Try all proofs with one step
- If none succeed, try all with two steps
- Next try all with three steps
- ...

Note: requires two important assumptions

- Proofs can be checked for correctness
- Only finitely many possible next steps

Entscheidungsproblem Fortsetzung

Claim: Entscheiwhatever is recursively enumerable

Proof:

- Try all proofs with one step
- If none succeed, try all with two steps
- Next try all with three steps
- ...

Note: requires two important assumptions

- Proofs can be checked for correctness
- Only finitely many possible next steps

Both true in sufficiently formal proof systems!

This All Seems Familiar...

Claim: If problem is RE, can reduce to Halting Prob

This All Seems Familiar...

Claim: If problem is RE, can reduce to Halting Prob

Proof:
Since RE, have "recognizer" `R`

This All Seems Familiar...

Claim: If problem is RE, can reduce to Halting Prob

Proof:
Since RE, have "recognizer" `R`
Suppose also have `TestHalt`

This All Seems Familiar...

Claim: If problem is RE, can reduce to Halting Prob

Proof:
Since RE, have "recognizer" `R`
Suppose also have `TestHalt`

```
Solver(x):
   if TestHalt(R, x) = false:
       return false
   else: return R(x)
```

This All Seems Familiar...

Claim: If problem is RE, can reduce to Halting Prob

Proof:
Since RE, have "recognizer" `R`
Suppose also have `TestHalt`

```
Solver(x):
    if TestHalt(R, x) = false:
        return false
    else: return R(x)
```

Used `TestHalt` to avoid problems if `R` loops!

Give Out Complements

What's so special about the false case?
What happens if we relax the true case instead?

[2] The co- stands for "complement"

Give Out Complements

What's so special about the false case?
What happens if we relax the true case instead?

Problem is co-RE[2] if $\exists$ program `P` st

- If answer for x is true, `P(x)` $=$ true *or loops*
- If answer for x is false, `P(x)` outputs false

[2] The co- stands for "complement"

Give Out Complements

What's so special about the false case?
What happens if we relax the true case instead?

Problem is co-RE[2] if $\exists$ program `P` st

- If answer for x is true, `P(x)` $=$ true *or loops*
- If answer for x is false, `P(x)` outputs false

Note: "opposite" of RE problem is co-RE
Ex: the "looping problem" is co-RE

[2] The co- stands for "complement"

Give Out Complements

What's so special about the false case?
What happens if we relax the true case instead?

Problem is co-RE[2] if $\exists$ program `P` st

- If answer for x is true, `P(x)` $=$ true *or loops*
- If answer for x is false, `P(x)` outputs false

Note: "opposite" of RE problem is co-RE
Ex: the "looping problem" is co-RE

Can RE problems be co-RE as well?

[2] The co- stands for "complement"

RE(EEEEEEEEEE)

Thm: Problem is RE and co-RE iff is computable

RE(EEEEEEEEEE)

Thm: Problem is RE and co-RE iff is computable

Proof (if):

- Solver satisfies both RE and co-RE

RE(EEEEEEEEEE)

Thm: Problem is RE and co-RE iff is computable

Proof (if):

- Solver satisfies both RE and co-RE

Proof (only if):

- Suppose have "recognizers" `R` and `CR`

RE(EEEEEEEEEE)

Thm: Problem is RE and co-RE iff is computable

Proof (if):

- Solver satisfies both RE and co-RE

Proof (only if):

- Suppose have "recognizers" `R` and `CR`
- Run `R(x)` and `CR(x)` in parallel
- Once one returns, use that answer

RE(EEEEEEEEEE)

Thm: Problem is RE and co-RE iff is computable

Proof (if):

- Solver satisfies both RE and co-RE

Proof (only if):

- Suppose have "recognizers" `R` and `CR`
- Run `R(x)` and `CR(x)` in parallel
- Once one returns, use that answer

Note: means halting not co-RE, looping not RE!

RE(EEEEEEEEEE)

Thm: Problem is RE and co-RE iff is computable

Proof (if):

- Solver satisfies both RE and co-RE

Proof (only if):

- Suppose have "recognizers" `R` and `CR`
- Run `R(x)` and `CR(x)` in parallel
- Once one returns, use that answer

Note: means halting not co-RE, looping not RE!

$\exists$ problems neither RE nor co-RE!
Beyond our scope though :'(

Fin

Next time: counting (with Elizabeth)!