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"

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
$S = \{x \mid x \notin x\}$

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

Yes?

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

No?

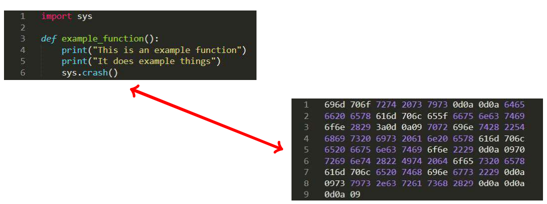
- If $S \notin S$, S defined to include S!

Set theory solution: make sure S not definable

In CS, not so easy to avoid!

An Important Aside

Computer programs $\equiv$ binary strings



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

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

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

Turing The Computer Scientist

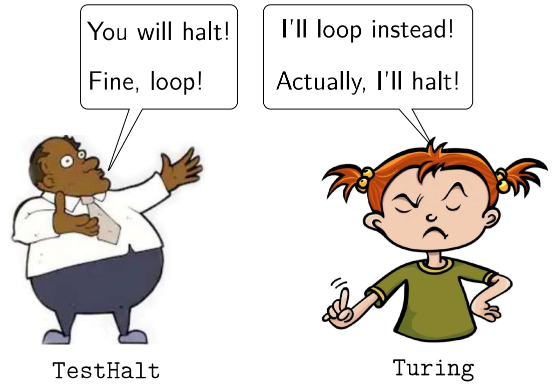
Assume for contradiction `TestHalt` exists

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

What does `Turing(Turing)` do?

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

Turing The Sassy Teenager



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

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')
```

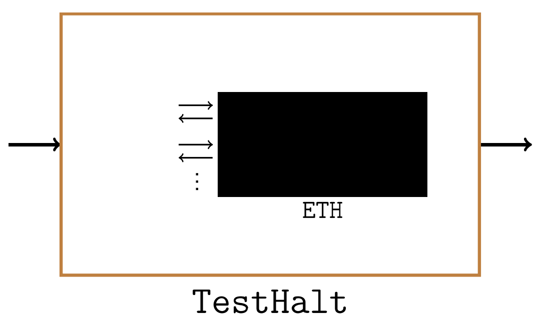
Input or none doesn't matter — can just hardcode!

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

Already Know `TestHalt` doesn't exist!

Reductionism

What we just did called a *reduction*



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?

Problem is *recursively enumerable*[1] if ∃ 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?

Claim: Entscheidungsproblem is undecidable

Proof:
Suppose ∃ 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

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

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?

Problem is co-RE[2] if ∃ 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

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!

∃ problems neither RE nor co-RE!
Beyond our scope though :'(

Fin

Next time: counting (with Elizabeth)!