

# Lecture 14: Computability

CS442: Great Insights in Computer Science  
Michael L. Littman, Spring 2006

## Today's Advice

- My only piece of advice for today:
- Don't listen to anything I say today;  
it will hurt your head.

# Self Contradiction

- The main topic today is self reference and self contradiction.
- The idea is that “interesting” things happen when something can refer to itself and assert that it has properties that negate its own existence.

# Oxymorons

- We’re all familiar with oxymorons: words that harbor two conflicting meanings.
- Top ten list from <http://www.oxymoronlist.com/>:

- |                             |                     |
|-----------------------------|---------------------|
| 20. Government Organization | 10. Pretty Ugly     |
| 19. Alone Together          | 9. Head Butt        |
| 18. Personal Computer       | 8. Working Vacation |
| 17. Silent Scream           | 7. Tax Return       |
| 16. Living Dead             | 6. Virtual Reality  |
| 15. Same Difference         | 5. Dodge Ram        |
| 14. Taped Live              | 4. Work Party       |
| 13. Plastic Glasses         | 3. Jumbo Shrimp     |
| 12. Tight Slacks            | 2. Healthy Tan      |
| 11. Peace Force             | 1. Microsoft Works  |

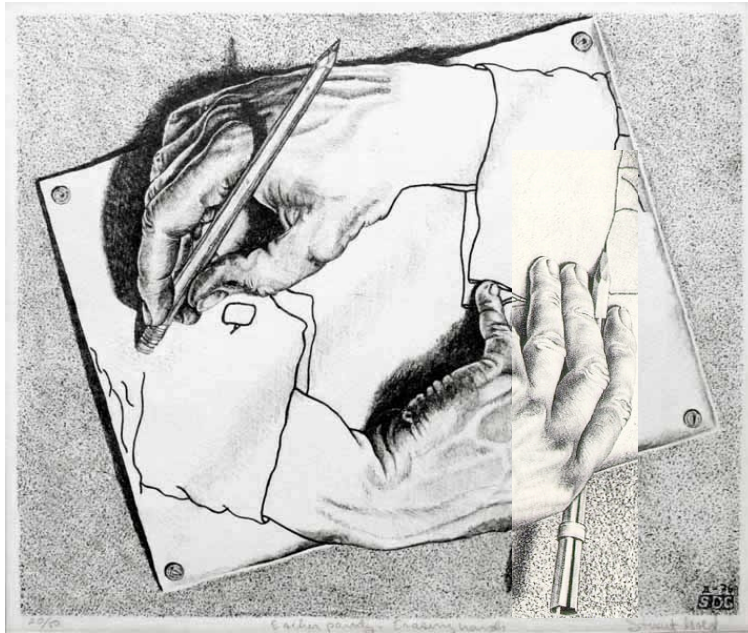
# Surface Contradiction

- Examples seem incongruous, but they all actually make sense.
- “jumbo shrimp” just means pretty big for a shrimp, which makes perfect sense. Like “pretty fly for a white guy”.
- The contradiction isn’t very deep.

# Deeper Self Denial

- Crowd: “We are all individuals!” Man: “I’m not.” (From *Life of Brian*)
- Eschew obfuscation.
- When you least expect it: expect it!
- “Last month I blew \$5,000 on a reincarnation seminar. I figured, hey, you only live once.”  
*Randy Shakes*

# Escher Parody



## Serious Fun

- There's a weird idea here: You shouldn't be able to create a statement that, if interpreted properly, results in another statement that contradicts the original statement.
- It's a "go back in time and kill your own grandfather" sort of thing.
- There are a bunch of deep mathematical insights that come from applying this idea.
- Here's a quick survey before the main event.

# Russell's Paradox

- In the town of Chelm, there's a barber. His job is to shave every man in town who does not shave himself.
- Who shaves the barber?
- For the statement to be true:
  - If he shaves himself, then he does not need to shave himself.
  - If he does not shave himself, then he needs to shave himself.



# Naive Set Theory

- This example shows that there are limitations to how you can create meaningful sets.
- If there is unrestricted self-reference, you can create impossible situations.

# Godel's Theorem

“This statement cannot be proven.”

- Kurt Godel showed that any system of mathematics that includes the integers can express this self-referential statement.
- If it's true, you can't prove it! (Incompleteness.)
- If it's false, you can prove something that's false! (Inconsistency.)
- Tough choice.

# Kantor's Diagonalization

- How many fractions are there? Infinite.
- How many decimals are there? Infinite.
- Are they the same size infinity?
- Well, we can make an infinitely long list that includes every fraction:

# List of Fractions

- Start with all the fractions where the numerator and denominator add up to 1, then 2, then 3.
  - Every fraction must eventually appear.
- |       |       |         |
|-------|-------|---------|
| $0/1$ | $1/5$ | $0/9$   |
| $0/2$ | $2/4$ | $1/8$   |
| $1/1$ | $3/3$ | $2/7$   |
| $0/3$ | $0/7$ | $3/6$   |
| $1/2$ | $1/6$ | $4/5$   |
| $0/4$ | $2/5$ | $0/10$  |
| $1/3$ | $3/4$ | $1/9$   |
| $2/2$ | $0/8$ | $2/8$   |
| $0/5$ | $1/7$ | $3/7$   |
| $1/4$ | $2/6$ | $4/6$   |
| $2/3$ | $3/5$ | $5/5$   |
| $0/6$ | $4/4$ | $\dots$ |

# And The Decimals?

- Can we list all the decimals?
- The “add to a constant” trick doesn’t work anymore, since we have decimals like  $0.3333\dots$  where the digit sum is infinite.
- So, let’s say we can list them all.
- Here’s the list, hypothetically:

# On The List?

- Read down the diagonal.  
 $0.5898032467\dots$
  - Add 1 to each digit (with wraparound).  
 $0.6909143578\dots$
  - The resulting decimal is not on the list!  
(Differs from the  $i$ th one in the  $i$ th digit.)
- |                       |
|-----------------------|
| $0.57953916570123654$ |
| $0.98877675309679680$ |
| $0.54921087722147810$ |
| $0.97889400202076116$ |
| $0.68930952976230064$ |
| $0.73758318399567813$ |
| $0.33201823212447767$ |
| $0.62085164445848273$ |
| $0.22859580612307950$ |
| $0.26912510570620329$ |
| $\dots$               |

# Conclusion

- You *can't* make a list of all the decimals.
- You *can* make a list of all the fractions.
- There are more decimals (real numbers) than fractions (rational numbers)!

# Looping Forever

- Given input 10, each of these programs counts backwards from 10 to 1.
- For each, is there an input  $n$  we can give that causes the program to loop forever?

```
def ex1(n):  
    for i in range(n,0,-1):  
        print i
```

```
def ex2(n):  
    while (n > 0):  
        print n  
        n = n - 1
```

```
def ex3(n):  
    while (1):  
        print n  
        n = n - 1  
        if (n == 0): return
```

# The Halting Problem

- Looping forever is one of the most annoying classes of programming errors.
- Would be great if a tool could automatically detect whether a program always halted.
- We'd like a subroutine **halt** that takes a program as input and returns *true* if the program halts on all inputs and *false* if some input makes it loop forever.

# Contrary

- If a **halt** subroutine exists, we can use it to create other programs.

```
def contrary(prog):  
    if prog == contrary:  
        if halt(prog) == true:  
            while (1):  
                print "loop"  
    return
```

- For example, this program takes a program “prog” as input, and, if prog is the program **contrary** and **halt(prog)** is true, **contrary** loops forever.
- Otherwise it halts.

# Contrary Analysis

- What does **contrary(contrary)** do?
- If **halt(contrary)** is true, that means **contrary** halts on all inputs, so it should halt on itself as input.
- But, in this case, **contrary(contrary)** loops forever!
- So, it must be that **halt(contrary)** is false, so **contrary** loops forever on some input.
- But, in this case, note that **contrary(prog)** halts for any input, including **contrary**.

# Halting Summary

- If **contrary(contrary)** halts, it loops forever.
- If **contrary(contrary)** loops forever, it halts.
- As with the Barber paradox, the problem here is our assumption, specifically, that **halt** exists.
- So, **halt** is a well-defined problem that no program can solve: It is *incomputable*.

# CS Implications

- There are many problems that turn out to be incomputable.
- All involve computations that might take an infinite number of comparisons to solve and you're never quite sure when to stop.
- An open problem I posed in my thesis (finding optimal policies for partially observable Markov decision processes) was later shown to be incomputable.

# Philosophical Implication

- Some have argued that since people can tell if programs halt but programs can't tell if programs halt, people are fundamentally more powerful / intelligent than computers.
- Hogwash.

# $3x+1$ Problem

- Take a number. Half it if it's even. Otherwise, triple and add 1. Continue until 1 is reached.
- Any power of 2 will be brought to 1 quickly.
- Some take awhile: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
- No one knows if it halts on all inputs!
- We can't (easily) solve the halting problem.

```
def even(x): return not x%2
```

```
def collatz(x):  
    while (x > 1):  
        print x  
        if even(x): x = x/2  
        else: x = 3*x+1
```

# Subber

- Here's an odd little aside.
- For many formal self-reference-based proofs, programs need to be able to refer to themselves.
- How do you do that?
- Consider a subroutine **subber**.
- It takes a string as input and produces a new string as output.
- The output is essentially a copy, but some special characters are converted (subbed).

# Code

- 2: “
  - 3: the whole string
  - 4: end of line
  - 5: tab
  - 6: “\n”
  - 7: “\t”
  - 8: “\\”
  - None of these characters are in **subber**, by the way.
- ```
def subber(q):
    o = ""
    for i in q:
        if i == str(1+1): o = o + ""
        elif i == str(1+1+1): o = o + q
        elif i == str(1+1+1+1): o = o + "\n"
        elif i == str(1+1+1+1+1): o = o + "\t"
        elif i == str(1+1+1+1+1+1): o = o + "\\ "+ "n"
        elif i == str(1+1+1+1+1+1+1): o = o + "\\ "+ "t"
        elif i == str(1+1+1+1+1+1+1+1): o = o + "\\ \\"
        else: o = o + i
    return o
```

# Self-Referential Program

- Running this program causes it to print precisely the program itself!

```
def selfPrint():  
    print subber("def selfPrint():45print subber(232)")
```

- Can even include **subber**, too:

```
def selfContained():  
    print subber("def subber(q):45o = 2245for i in q:455if i == str(1+1): o = o +  
'2'455elif i == str(1+1+1): o = o + q455elif i == str(1+1+1+1): o = o + 262455elif i ==  
str(1+1+1+1+1): o = o + 272455elif i == str(1+1+1+1+1+1): o = o + 282+2n2455elif i  
== str(1+1+1+1+1+1+1): o = o + 282+2t2455elif i == str(1+1+1+1+1+1+1+1): o = o +  
2882455else: o = o + i45return o44def selfContained():45print subber(232)")
```

## Weird?

- Something weird and “birth”-like here. The program has a string, which is the program, which somehow has the string, which is the program...
- Should be infinitely big, but it’s not via clever use of variables and substitution.

# Next Time

- Cryptography.
- Finish Hillis, Chapter 4.