6.001 SICP
Computability

- What we've seen...

- Deep question #1:
  - Does every expression stand for a value?

- Deep question #2:
  - Are there things we *can't* compute?

- Deep question #3:
  - What is EVAL really?

(1) Abstraction

- Elements of a Language

- **Procedural** Abstraction:
  - Lambda – captures common patterns and "how to" knowledge

- Functional programming & substitution model

- Conventional interfaces:
  - list-oriented programming
  - higher order procedures

(2) Data, State and Objects

- **Data** Abstraction
  - Primitive, Compound, & Symbolic Data
  - Contracts, Abstract Data Types
  - Selectors, constructors, operators, ...

- Mutation: need for environment model

- Managing complexity
  - modularity
  - data directed programming
  - object oriented programming

(3) Language Design and Implementation

- Evaluation – meta-circular evaluator
  - eval & apply

- Language extensions & design
  - lazy evaluation; streams
  - dynamic scoping

- Register machines
  - ec-eval

Time for some prizes!

Deep Question #1

Does every expression stand for a value?

Some Simple Procedures

- Consider the following procedures

```
(define (return-seven) 7)
(define (loop-forever) (loop-forever))
```

- So

```
(return-seven)
```

⇒ **7**

```
(loop-forever)
```

⇒ [never returns!]

- Expression **`(loop-forever)`** does not stand for a value; not well defined.

Deep Question #2

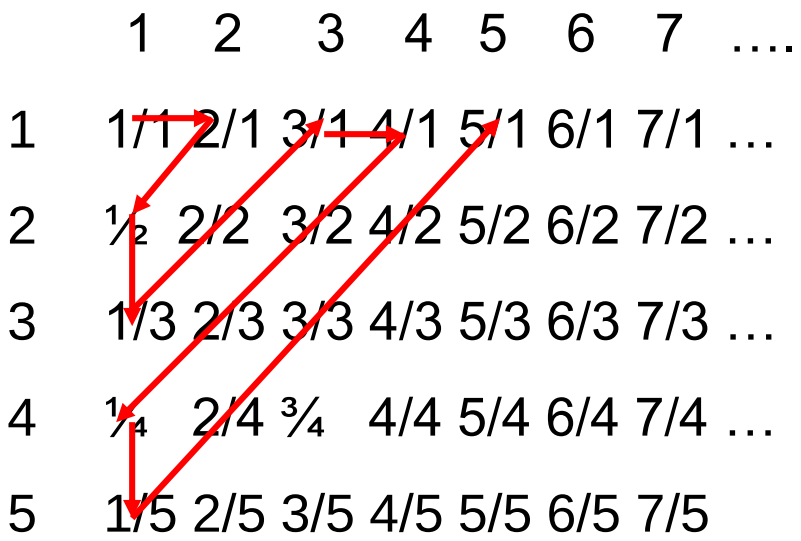
Are there well-defined things that cannot be computed?

Countable and uncountable

- Two sets of numbers (or other objects) are said to have the same cardinality (or size) if there is a one-to-one mapping between them. This means each element in the first set matches to exactly one element in the second set, and vice versa.
- Any set of same cardinality as the integers is called countable.
- {integers} maps to {even integers}: n → 2n
- {integers} maps to {squares}: n→n^2
- {integers} maps to {rational fractions between 0 and 1}

Countable and uncountable

- Proof of last claim



- Mapping from this set to integers – count from 1 as move along line
- Mapping from integers to this set – first row clearly contains integers

Countable and uncountable

- The set of numbers between 0 and 1 is uncountable, I.e. there are more of them than there are integers:
- Represent any such number by binary fraction, e.g. 0.01011 → ¼ + 1/16 + 1/32
- Assume there are a countable number of such numbers. Then can arbitrarily number them, as in this table:

	½	¼	1/8	1/16	1/32	1/64	….
1	0	1	0	1	1	0	….
2	1	1	0	1	0	1	….
3	0	0	1	0	1	0	….

• Pick a new number by complementing the diagonal, e.g.100… This number cannot be in the list!! So the assumption of countability is false, and there are more irrationals than rationals

There are more functions than programs

- There are clearly a countable number of procedures, since each is of finite length, and based on a finite alphabet.
- Assume there are a countable number of predicate functions, i.e. mappings from an integer argument to the values 0 or 1. Then we can arbitrarily number them.

	1	2	3	4	5	6	….
P1	**0**	1	0	1	1	0	….
P2	1	**1**	0	1	0	1	….
P3	0	0	**1**	0	1	0	….

• Play the same Cantor Diagonalization game. Define a new predicate function by complementing the diagonals. By construction this predicate cannot be in the list, yet we claimed we could list all of them. Thus there are more predicate functions than there are procedures.

halts?

- Even our simple procedures can cause trouble. Suppose we wanted to check procedures before running them to catch accidental infinite loops.

- Assume a procedure **`halts?`** exists:

  `(halts? p)`

  ⇒ **`#t if (p) terminates`**

  ⇒ **`#f if (p) does not terminate`**

- **`halts?`** is well specified – has a clear value for its inputs

  `(halts? return-seven)` ➔ **`#t`**

  `(halts? loop-forever)` ➔ **`#f`**

Halperin, Kaiser, and Knight, "Concrete Abstractions," p. 114, ITP 1999.

The Halting Theorem:
Procedure `halts?` cannot exist. Too bad!

- Proof (informal): Assume **`halts?`** exists as specified.

```
(define (contradict-halts)
  (if (halts? contradict-halts)
      (loop-forever)
      #t))

(contradict-halts)
⇒ ??????
```

- Wow!
  - If **`contradict-halts`** halts, then it loops forever.
  - If **`contradict-halts`** doesn't halt, then it halts.
- Contradiction!
  Assumption that **`halts?`** exists must be wrong.

Deep Question #3

What is EVAL really?

The dream of a universal machine…

The dream of a universal machine

A bright red Ferrari F-430…

ACME Universal machine

If you can say it, I can do it™

What is Eval really?

- We *do* describe devices, in a language called Scheme
- We have a machine that takes those descriptions and then behaves *exactly* as they specify
- Eval takes *any program* as input and reconfigures itself to simulate that input program
- EVAL *is* a *universal machine*

Seen another way

- Suppose you were a circuit designer
  - Given a circuit diagram, you could transform it into an electric signal encoding the layout of the diagram
  - Now suppose you wanted to build a circuit that could take any such signal as input (any other circuit) and could then reconfigure itself to simulate that input circuit
  - What would this general circuit look like???
- Suppose instead you describe a circuit as a program
  - Can you build a program that takes any program as input and reconfigures itself to simulate that input program?
  - Sure – that's just EVAL!! – it's a UNIVERSAL MACHINE

It wasn't always this obvious

- "If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered"

Howard Aiken, writing in 1956 (designer of the Mark I "Electronic Brain", developed jointly by IBM and Harvard starting in 1939

Why a Universal Machine?

- If EVAL can simulate any machine, and if EVAL is itself a description of a machine, then EVAL can simulate itself
  - This was our example of *meval*
- In fact, EVAL can simulate an evaluator for any other language
  - Just need to specify syntax, rules of evaluation
- An evaluator for any language can simulate any other language
  - Hence there is a general notion of computability – idea that a process can be computed independent of what language we are using, and that anything computable in one language is computable in any other language

Turing’s insight

- Alan Mathison Turing
- 1912-1954

Turing's insight

- Was fascinated by Godel's incompleteness results in decidability (1933)
  - In any axiomatic mathematical system there are propositions that cannot be proved or disproved within the axioms of the system
  - In particular the consistency of the axioms cannot be proved.
- Led Turing to investigate Hilbert's Entscheidungsproblem
  - Given a mathematical proposition could one find an algorithm which would decide if the proposition was true or false?
  - For many propositions it was easy to find such an algorithm.
  - The real difficulty arose in proving that for certain propositions no such algorithm existed.
  - In general – Is there some fixed definite process which, in principle, can answer any mathematical question?
  - E.g., Suppose want to prove some theorem in geometry
    - Consider all proofs from axioms in 1 step
    - … in 2 steps ….

Turing's insight

- Turing proposed a theoretical model of a simple kind of machine (now called a Turing machine) and argued that any "effective process" can be carried out by such a machine
  - Each machine can be characterized by its program
  - Programs can be coded and used as input to a machine
  - Showed how to code a universal machine
  - Wrote the first EVAL!

The halting problem

- If there is a problem that the universal machine can't solve, then no machine can solve, and hence no effective process
- Make list of all possible programs (all machines with 1 input)
- Encode all their possible inputs as integers
- List their outputs for all possible inputs (as integer, error or loops forever)
- Define f(n) = output of machine n on input n, plus 1 if output is a number
- Define f(n) = 0 if machine n on input n is error or loops
- But f can't be computed by any program in the list!!
- Yet we just described process for computing f??
- Bug is that can't tell if a machine will always halt and produce an answer

The Halting theorem

- Halting problem: Take as inputs the description of a machine M and a number n, and determine whether or not M will halt and produce an answer when given n as an input
- Halting theorem (Turing): There is no way to write a program (for any computer, in any language) that solves the halting problem.

Turing's history

- Published this work as a student
  - Got exactly two requests for reprints
  - One from Alonzo Church (professor of logic at Princeton)
    - Had his own formalism for notion of an effective procedure, called the lambda calculus
- Completed Ph.D. with Church, proving Church-Turing Thesis:
  - Any procedure that could reasonably be considered to be an effective procedure can be carried out by a universal machine (and therefore by any universal machine)

Turing's history

- Worked as code breaker during WWII
  - Key person in Ultra project, breaking German's Enigma coding machine
  - Designed and built the *Bombe,* machine for breaking messages from German Airforce
  - Designed statistical methods for breaking messages from German Navy
  - Spent considerable time determining counter measures for providing alternative sources of information so Germans wouldn't know Enigma broken
  - Designed general-purpose digital computer based on this work
- Turing test: argued that intelligence can be described by an effective procedure – foundation for AI
- World class marathoner – fifth in Olympic qualifying (2:46:03 – 10 minutes off Olympic pace)
- Working on computational biology – how nature "computes" biological forms.
- His death

Good luck on the final!!

Deep Question #3

Where does the power of recursion come from?

From Whence Recursion?

- Perhaps the ability comes from the ability to DEFINE a procedure and call that procedure from within itself?

  Consider the infinite loop as the purest or simplest invocation of recursion:

```
(define (loop) (loop))
```

- Can we generate recursion without DEFINE (i.e. is something other than DEFINE at the heart of recursion)?

Infinite Recursion without Define

- We have notion of lambda, which abstracts out the pattern of computation and parameterizes that computation. Perhaps try:

```
((lambda (loop) (loop))
 (lambda (loop) (loop)))
```

- Not quite: problem is that loop requires one argument, and the first application is okay, but the second one isn't:

⇒`((lambda (loop) (loop)) ____ )` ; missing arg

Infinite Recursion without Define

- Better is ....

```
((λ(h) (h h)) ;
 (λ(h) (h h)))
```

an anonymous infinite loop!

- Run the substitution model:

```
  ((λ(h) (h h))
   (λ(h) (h h)))
= (H H)
⇒(H H)
⇒(H H)
   ...
```

H (shorthand)

- Generate infinite recursion with only **lambda** and **apply**.

Harnessing recursion

- Cute but so what?

- How is it that we are able to compute many (interesting) things, e.g.

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

- Can compute factorial for any finite positive integer n (given enough, but finite, memory and time)

Harness this anonymous recursion?

- We'd like to do something each time we recurse:

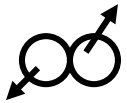
```
((λ(h) (f (h h)))
 (λ(h) (f (h h))))
= (Q Q)
⇒ (f (Q Q))
⇒ (f (f (Q Q)))
⇒ (f (f (f ... (f (Q Q))..)
```

- So our first step in harnessing recursion still results in infinite recursion... but at least it generates the "stack up" of **f** as we expect in recursion

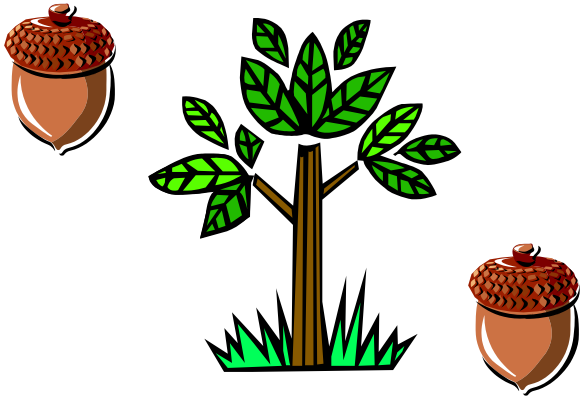
How do we stop the recursion?

- We need to subdue the infinite recursion – how to prevent `(Q Q)` from spinning out of control?

```
   ((λ(h) (λ(x) ((f (h h)) x)))
    (λ(h) (λ(x) ((f (h h)) x))))
=  (D D)
⇒(λ(x) ((f (D D)) x))
⇒
```



```
      p: x
      b: ((f (D D)) x)
```

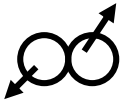


- So `(D D)` results in something very finite – a procedure!
- That procedure object has the germ or seed `(D D)` inside it – the potential for further recursion!

Compare

```
(Q Q)
⇒ (f (f (f ... (f (Q Q))..)
```

- (Q Q) is **uncontrolled** by `f`; it evals to itself by itself

```
(D D)
⇒ (λ(x) ((f (D D)) x))
⇒ 
    p: x
    b: ((f (D D)) x)
```

- `(D D)` temporarily halts the recursion and gives us mechanism to **control** that recursion:
  1. trigger proc body by applying it to number
  2. Let `f` decide what to do – call other procedures

Parameterize (capture f)

- In our funky recursive form `(D D)`, `f` is a free variable:

```
((λ(h) (λ(x) ((f (h h)) x)))
 (λ(h) (λ(x) ((f (h h)) x))))
= (D D)
```

- Can clean this up: formally parameterize what we have so it can take `f` as a variable:

```
(λ(f) ((λ(h) (λ(x) ((f (h h)) x)))
       (λ(h) (λ(x) ((f (h h)) x)))))
```

= **Y**

The Y Combinator

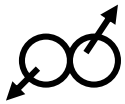
```
(λ(f) ((λ(h) (λ(x) ((f (h h)) x)))
       (λ(h) (λ(x) ((f (h h)) x)))))
```

= **Y**

- So

```
(Y F) = (D D)
```

⇒

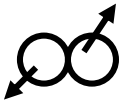


```
p: x
b: ((F (D D)) x)
```

as before, where **f** is bound to some form **F**. That is to say, when we use the **Y** combinator on a procedure **F**, we get the controlled recursive capability of **(D D)** we saw earlier.

How to Design F to Work with Y?

```
(Y F) = (D D)
```

⇒ 

```
p: x
b: ((F (D D)) x)
```

- Want to design **F** so that we control the recursion. What form should **F** take?
- When we feed **(Y F)** a number, what happens?

```
((Y F) #)
```

⇒ ( #)

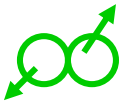
```
p: x
b: ((F (D D)) x)
```

⇒ ((F ) #)

```
p: x
b: ((F (D D)) x)
```

1. **F** should take a proc
2. **(F proc)** should eval to a procedure that takes a number

Implication of 2: F Can End the Recursion

⇒ ((F ) #)

p: x
b: ((F (D D)) x)

```
F = (λ(proc)
     (λ(n)
      ...))
```

- Can use this to complete a computation, depending on value of n:

```
F = (λ(proc)
     (λ(n)
      (if (= n 0)
          1
          ...)))
```

Let's try it!

Example: An F That Terminates a Recursion

```
F = (λ(proc)
      (λ(n) (if (= n 0) 1 ...)))
```

So

```
((F  ) 0)
```

```
p: x
b: ((F (D D)) x)
```

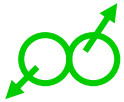
```
⇒ ((λ(n) (if (= n 0) 1 ...)) 0)
⇒ 1
```

- If we write **F** to bottom out for some values of n, we can implement a base case!

Implication of 1: F Should have Proc as Arg

- The more complicated (confusing) issue is how to arrange for F to take a proc of the form we need:

We need F to conform to:

```
((F  ) 0)
```

```
p: x
b: ((F (D D)) x)
```

- Imagine that F uses this proc somewhere inside itself

```
F = (λ(proc)
     (λ(n)
      (if (= n 0) 1 ... (proc #) ...)))
  = (λ(proc)
     (λ(n)
      (if (= n 0) 1 ... (  #) ...)))
```

```
p: x
b: ((F (D D)) x)
```

Implication of 1: F Should have Proc as Arg

- Question is: how do we appropriately use proc inside **F**?
- Well, when we use proc, what happens?

```
( ↺ #)
p: x
b: ((F (D D)) x)

⇒ ((F (D D)) #)
⇒ ((F ↺ ) #)
      p: x
      b: ((F (D D)) x)

⇒ ((λ(n) (if (= n 0) 1 ...))  #)
⇒ (if (= # 0) 1 ...)
```

Wow! We get the eval of the inner body of F with n=#

Implication of 1: F Should have Proc as Arg

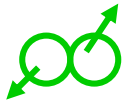
- Let's repeat that:

```
(proc #)
```
-- when called inside the body of F

⇒ `( `  `#)`

```
p: x
b: ((F (D D)) x)
```

⇒ is just the inner body of F with n = #, **and proc =**



```
p: x
b: ((F (D D)) x)
```

- So consider

```
F = (λ(proc)
     (λ(n)
      (if (= n 0)
          1
          (* n (proc (- n 1))))))
```

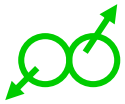
So What is proc?

- Consider our procedure

```
F = (λ(proc)
     (λ(n)
      (if (= n 0)
          1
          (* n (proc (- n 1))))))
```

- This is pretty wild! It requires a very complicated form for proc in order for everything to work recursively as desired.
- How do we get this complicated proc? **Y** makes it for us!

```
(Y F) = (D D) =>
```



```
= proc
p: x
b: ((F (D D)) x)
```

Putting it all together

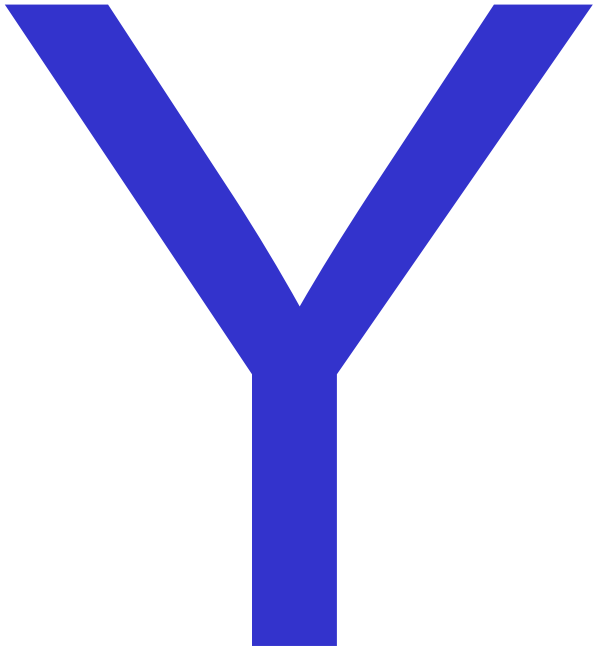
```
( (Y F) 10)  =
( ((λ(f) ((λ(h) (λ(x) ((f (h h)) x)))
          (λ(h) (λ(x) ((f (h h)) x)))))
   (λ(fact)
     (λ(n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))))
   10)

⇒ (* 10 (* ... (* 3 (* 2 (* 1 1)))
⇒ 3628800
```

Y Combinator: The Essence of Recursion

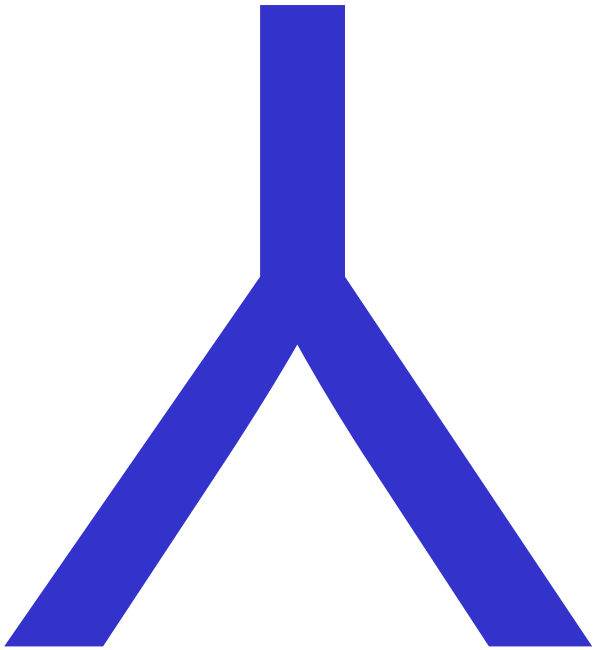
```
((Y F) x) = ((D D) x) = ((F (Y F)) x)
```

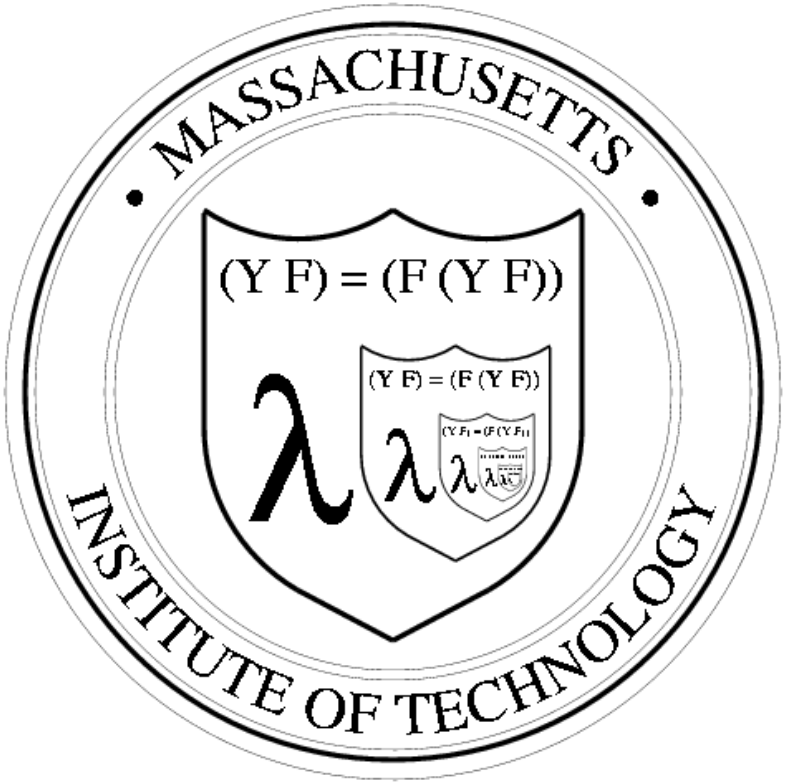
The power of controlled recursion!



The Limits of Lambda and Y

- We can approximate infinity, but not quite reach it...
- Y gives us the power to reach toward and control infinite recursion one step at a time;
- But there are limits – remember the halting theorem!




MASSACHUSETTS
(Y F) = (F (Y F))
INSTITUTE OF TECHNOLOGY