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:
  - Where does our computational power (of recursion) come from?

(1) Abstraction

- Elements of a Language (or Engineering Design)
  - Primitives, means of combination, means of abstraction
- **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
  - dynamic scoping
- Register machines
  - ec-eval and universal machines
  - compilation
  - list structured data and memory management

Deep Question #1

Does every expression stand for a value?

Some Simple Procedures

- Consider the following procedures

```
(define (return-seven) (+ 3 4))
(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?

Mysteries of Infinity: Countability

- 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} same size as {even integers}: n → 2n
- {integers} same size as {squares}: $n \rightarrow n^2$
- {integers} same size as {rational numbers}

Countable – rational numbers

- As many integers as rational numbers (no more, no less). Proof:

	1	2	3	4	5	6	7 …
1	1/1	2/1	3/1	4/1	5/1	6/1	7/1 …
2	1/2	2/2	3/2	4/2	5/2	6/2	7/2 …
3	1/3	2/3	3/3	4/3	5/3	6/3	7/3 …
4	1/4	2/4	3/4	4/4	5/4	6/4	7/4 …
5	1/5	2/5	3/5	4/5	5/5	6/5	7/5 …

- Mapping between the set of rationals and set of integers – match integers to rationals starting from 1 as move along line

Uncountable – real numbers

- The set of real numbers between 0 and 1 is uncountable, i.e. there are more of them than there are integers:
- Proof: Represent a real number by its decimal expansion (may require an infinite number of digits), e.g. 0.49373
- Assume there are a countable number of such numbers. Then can arbitrarily number them, as in this table:

#1 0. 4 9 3 7 3 0 0 0 ...
#2 0. 3 3 3 3 3 3 3 3 ...
#3 0. 5 8 7 5 3 2 1 4 ...

- Pick a new number by adding 1 (modulo 10) to every element on the diagonal, e.g. 0.437... becomes 0.548… This number cannot be in the list! The assumption of countability is false, and there are more reals than integers

There are more functions than programs

- There are a countable number of procedures: e.g. can write every program in a binary (integer) form, 100110100110
- Assume there are a countable number of predicate functions, i.e. mappings from an integer arg to the value 0 or 1. Then we can arbitrarily number these functions:

#1 0 1 0 1 1 0 …
#2 1 1 0 1 0 1 …
#3 0 0 1 0 1 0 …

- Use Cantor Diagonalization again! Define a new predicate function by complementing the diagonals. By construction this predicate cannot be in the list (of all integers, of all programs). Thus there are more predicate functions than there are procedures.

halts?

- Even simple procedures can cause deep difficulties. 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.
- Contradiction!
  Assumption that **halts?** exists must be wrong.

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?
- Example: 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 the *power to name* 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)) ; **an anonymous infinite loop!**
(λ(h) (h h)))

- Run the substitution model:

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

H (shorthand)

- Can generate infinite recursion with only **lambda** & **apply**

Harnessing recursion

- So lambda (not naming) gives us recursion. But do we still need the power to name (define) in order to do anything practical or useful?
- For example, computing factorials:

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

- Can we compute factorials without explicitly "naming" such a procedure?

Harnessing our anonymous recursion

```
((λ(h) (h h)) ; our anonymous infinite loop
 (λ(h) (h h)))
```

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

```
((λ(h) (f (h h)))     Q (shorthand)
 (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 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 an argument:

```
(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, but now **f** is bound to some form **F**. 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!

25

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!

26

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)
```

27

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 ...)
```

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

28

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))))))
```

29

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)
```

30

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
```

No **`define`** – only **`lambda`** and the power of Y combinator!

31

Y Combinator: The Essence of Recursion

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

The power of controlled recursion!

Y

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

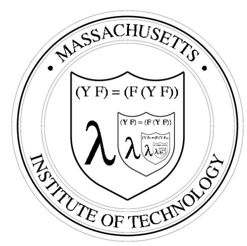
32

The Power and Its Limits

- λ empowers you to capture knowledge
- Y empowers you to reach toward the infinite – to control infinite recursion one step at a time
- But there are limits – remember the halting theorem!

Y ⅄

33



34