

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?

1/44

(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

2/44

(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

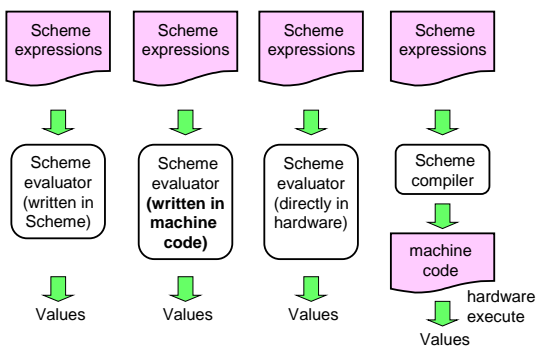
3/44

(3) Language Design and Implementation

- Evaluation – meta-circular evaluator
 - eval & apply
- Language extensions & design
 - lazy evaluation
 - dynamic scoping
 - wild ideas, e.g. streams

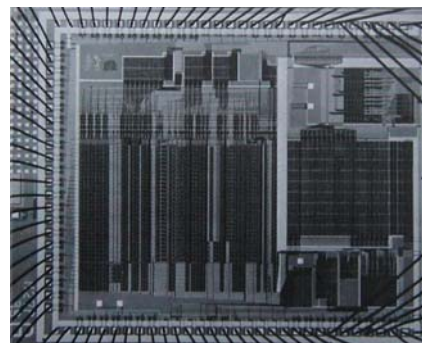
4/44

Implementations of Scheme



5/44

Silicon Chip Implementation of a Scheme Evaluator



SICP, p. 548

6/44

Syntax and Semantics

- Syntax: structure
- Semantics: meaning
- In English
 - syntax: the structure of a sentence
 - The dark blue box fell quickly.
 - semantics: what that sentence means
- In Scheme

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

 - what *value* does (fact 10) produce?
- The colorless green ideas slept furiously.
- Are there syntactically valid but meaningless scheme programs?

7/44

Deep Question #1

Does every expression stand for a value?

8/44

Some Simple Procedures

- Consider the following procedures

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

```
(return-seven)
⇒ 7

(loop-forever)
⇒ [never returns!]
```

9/44

Deep Question #2

Are there well-defined things that cannot be computed?

10/44

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 natural numbers (non-negative integers) is called **countable**.

11/44

Countable – rational numbers

- Proof of claim

	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	...

- Unique one-to-one mapping from this set to non-negative integers – count from 1 as move along zig-zag line

12/44

Uncountable – real numbers

- The set of numbers between 0 and 1 is uncountable, i.e. there are more of them than there are natural numbers:
 - represent any such number by binary fraction, e.g. $0.01011 \rightarrow 1/4 + 1/16 + 1/32$
- Assume there are a countable number of such numbers. Then can arbitrarily number them, as in this table:

	1/2	1/4	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 thus there are more real numbers than rationals

13/44

Timeout: Scheme Trivial Pursuit

- What's a function?
- What's a procedure/program?
- Not so trivial observation: there are more *functions* than there are *procedures/programs*.

14/44

There are more functions than programs

- There are a countable number of procedures: finite length, finite alphabet.
- Consider only *predicate functions* with a single integer argument and a 0 or 1 answer. (Clearly a subset of *all functions*.) Assume there are a countable number of these.
- Then we can number them and match each up to a procedure. The table shows the output of each function (row) on each of the integers (col).

	i1	i2	i3	i4	i5	i6	...
P1/f1	0	1	0	1	1	0	...
P2/f2	1	1	0	1	0	1	...
P3/f3	0	0	1	0	1	0	...

Play the same Cantor Diagonalization game: Define a new function by complementing the diagonals. This predicate cannot be in the list of procedures, yet we said we could list all of them.

Thus there are more predicate functions than there are procedures.

15/44

halts?

- Even our simple procedures can cause trouble.
- Assume a procedure `halts?` exists:
 - `(halts? p)`
 - \Rightarrow #t if (p) terminates
 - \Rightarrow #f if (p) does not terminate
- `halts?` is well specified – has a clear value for its inputs
 - `(halts? return-seven) \rightarrow #t`
 - `(halts? loop-forever) \rightarrow #f`

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

16/44

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)
 $\Rightarrow$  ???????
```

- If `contradict-halts` halts, then it loops forever.
- Assumption that `halts?` exists must be wrong.

17/44

Some we can't, but some we can...

So:

- There are some well specified things we cannot compute

But...

- There are also interesting things we CAN compute
 - Many of our most interesting (powerful) procedures have been *recursive*

18/44

Deep Question #3

Where does the power of recursion come from?

20/44

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)?

21/44

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

H (shorthand)

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

23/44

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)

24/44

Harness this anonymous recursion?

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

```
((λ(h) (display "hi!") (h h))  
 (λ(h) (display "hi!") (h h)))
```

- Somewhat more interesting: `((λ(h) (f (h h))) (λ(h) (f (h h))))` Q (shorthand)

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

25/44

How do we stop the recursion?

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

– Previously: `(λ(h) (f (h h)))`

– Now: `((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!

26/44

Compare

(Q Q)
 $\Rightarrow (F (F (F \dots (F (Q Q)) \dots))$
 • (Q Q) is **uncontrolled** by **F**; it evals to itself by itself

(D D)
 $\Rightarrow (\lambda(x) ((F (D D)) x))$ – *proc with special structure!*

\Rightarrow 
 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

27/44

Parameterize (capture f)

- In our funky recursive form (D D), **F** is a free variable:

$((\lambda(h) (\lambda(x) ((F (h h)) x)))$
 $(\lambda(h) (\lambda(x) ((F (h h)) x))))$

= (D D)

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

$(\lambda(F) ((\lambda(h) (\lambda(x) ((F (h h)) x)))$
 $(\lambda(h) (\lambda(x) ((F (h h)) x))))$)


= **Y**

28/44

The Y Combinator

$(\lambda(F) ((\lambda(h) (\lambda(x) ((F (h h)) x)))$
 $(\lambda(h) (\lambda(x) ((F (h h)) x))))$)


= **Y**

• So
 $(Y F) = (D D)$
 \Rightarrow 
 p: x
 b: ((F (D D)) x)

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.

29/44


How to Design F to Work with Y?

$(Y F) = (D D)$
 \Rightarrow 
 p: x
 b: ((F (D D)) x)



- Want to design **F** so that we control the recursion. What form should **F** take?
 - **F** should take a **proc**

30/44

How to Design F to Work with Y?

$(Y F) = (D D)$
 \Rightarrow 
 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?


$((Y F) \#)$
 \Rightarrow ( #) (**F proc**) should eval to a procedure that takes a number
 p: x
 b: ((F (D D)) x)
 \Rightarrow (() #)
 p: x
 b: ((F (D D)) x)

31/44

So What is proc?

• **F** = $(\lambda(\text{proc})$
 $(\lambda(n)$
 $(\text{if} (= n 0)$
 1
 $(* n (\text{proc} (- n 1))))))$

- This is pretty wild! It requires a very complicated form for **proc** in order for everything to work recursively as desired.
- In essence, **proc** encapsulates the factorial computation itself, without naming it explicitly
- How do we get this complicated **proc**? **Y** makes it for us!

$(Y F) = (D D) \Rightarrow$  = **proc**
 p: x
 b: ((F (D D)) x)

32/44

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(g)
    (λ(n)
      (if (= n 0)
            1
            (* n (g (- n 1))))))
  10)
⇒ (* 10 (* ... (* 3 (* 2 (* 1 1)))
⇒ 3628800

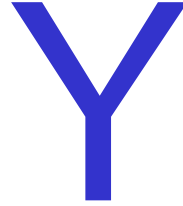
```

33/44

Y Combinator: The Essence of Recursion

$((Y F) x) = ((D D) x) = ((F (Y F)) x)$

The power of controlled recursion!



34/44

Lambda and Y

- Lambda gives you the power to capture knowledge
- Y gives you the power to reach toward and control infinite recursion one step at a time;
- There are limits:
 - we can approximate infinity, but not quite reach it... remember the halting theorem!



35/44



36/44

Appendix: Details of how to design **proc**

- The following slides give the details of how you might figure out what the structure must be of the procedure passed to Y that will cause Y to create the recursion that computes factorial.
- We have seen similar, though simpler examples of how to design programs from an understanding of the constraints imposed by what their input and output types must be.

37/44

Implication of 2: F Can End the Recursion

⇒ ((**F** ) #)

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

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

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

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

```

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

```

38/44

Example: An F That Terminates a Recursion

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

So

```
((F (proc) ) 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!

39/44

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 (proc) ) 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 ... ((F (D D)) #) ...)))
  p: x
  b: ((F (D D)) x)
```

40/44

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?

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

```
⇒ ((F (D D)) #)
```

```
⇒ ((F (proc) #)
```

```
  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=#

41/44

Implication of 1: F Should have Proc as Arg

- Let's repeat that:

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

```
⇒ ((F (proc) #)
```

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

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

```
(proc)
F = (λ(proc)
      (λ(n)
        (if (= n 0)
            1
            (* n (proc (- n 1)))))))
  p: x
  b: ((F (D D)) x)
```

42/44

More Ideas – Nondeterministic Computing

The following puzzle (taken from Dinesman 1968) is typical of a large class of simple logic puzzles:

Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

43/44

More Ideas – Nondeterministic Computing

Can extend our language/interpreter to "solve" such problems! (SICP §4.3)

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker) (list 'cooper cooper)
          (list 'fletcher fletcher) (list 'miller miller)
          (list 'smith smith))))
```

Evaluating the expression (multiple-dwelling) produces the result
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))

44/44