

Basic Scheme

February 8, 2007

- Compound expressions
- Rules of evaluation
- Creating procedures by capturing common patterns

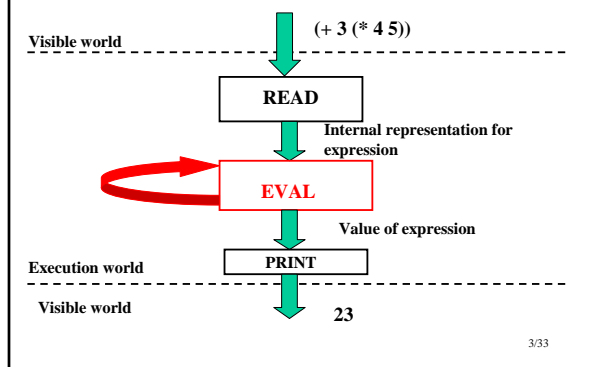
1/33

Previous lecture

- Basics of Scheme
 - Expressions and associated values (or syntax and semantics)
 - Self-evaluating expressions
 - 1, "this is a string", #f
 - Names
 - +, *, >=, <
 - Combinations
 - (* (+ 1 2) 3)
 - Define
- Rules for evaluation

2/33

Read-Eval-Print



3/33

Summary of expressions

- **Numbers:** value is expression itself
- **Primitive procedure names:** value is pointer to internal hardware to perform operation
- **"Define":** has no actual value; is used to create a **binding in a table** of a name and a value
- **Names:** value is looked up in table, retrieving binding
- Rules apply recursively

4/33

Simple examples

25	→	25
(+ (* 3 5) 4)	→	19
+	→	[#primitive procedure ...]
(define foobar (* 3 5))	→	no value, creates binding of foobar and 15
foobar	→	15 (value is looked up)
(define fred +)	→	no value, creates binding
(fred 3 5)	→	8

5/33

This lecture

Adding procedures and procedural abstractions to capture processes

6/33

Language elements -- procedures

- Need to capture ways of doing things – use procedures

`(lambda (x) (* x x))`

↑ parameters ↑ body

To process something multiply it by itself



- Special form – creates a procedure and returns it as value

7/33

Language elements -- procedures

- Use this anywhere you would use a procedure

`((lambda(x) (* x x)) 5)`

← lambda exp ← arg

25

8/33

Language elements -- abstraction

- Use this anywhere you would use a procedure

`((lambda(x) (* x x)) 5)`

Don't want to have to write obfuscatory code – so can give the lambda a name

`(define square (lambda (x) (* x x)))` **Rumplestiltskin effect!**
`(square 5) → 25` *(The power of naming things)*

9/33

Scheme Basics

- Rules for *evaluating*
 - If **self-evaluating**, return value.
 - If a **name**, return value associated with name in environment.
 - If a **special form**, do something special.
 - If a **combination**, then
 - Evaluate all of the subexpressions of combination (in any order)
 - apply the operator to the values of the operands (arguments) and return result
- Rules for *applying*
 - If procedure is **primitive procedure**, just do it.
 - If procedure is a **compound procedure**, then: evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

10/33

Interaction of define and lambda

```
1. (lambda (x) (* x x))
   ==> #[compound-procedure 9]
2. (define square (lambda (x) (* x x)))
   ==> undef
3. (square 4)
   ==> 16
4. ((lambda (x) (* x x)) 4)
   ==> 16
5. (define (square x) (* x x)) ==> undef
```

This is a convenient shorthand (called “syntactic sugar”) for 2 above – this is a use of lambda!

11/33

Lambda special form

- lambda syntax `(lambda (x y) (/ (+ x y) 2))`
- 1st operand position: the **parameter list** `(x y)`
 - a list of names (perhaps empty) `()`
 - determines the number of operands required
- 2nd operand position: the **body** `(/ (+ x y) 2)`
 - may be any expression(s)
 - not evaluated when the lambda is evaluated
 - evaluated when the procedure is applied
 - value of body is value of last expression evaluated
- mini-quiz: `(define x (lambda () (+ 3 2)))`
 - `x`
 - `(x)`
- semantics of lambda:

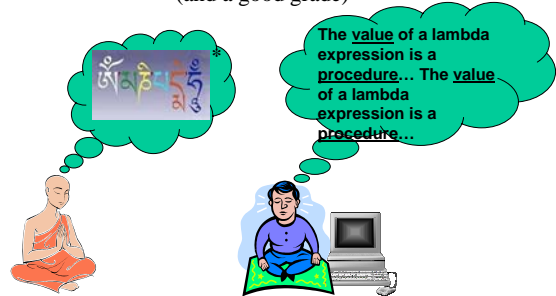
12/33

THE VALUE OF A LAMBDA EXPRESSION IS A PROCEDURE

13/33

Achieving Inner Peace

(and a good grade)



*Om Mani Padme Hum...

14/33

Using procedures to describe processes

- How can we use the idea of a procedure to capture a computational process?

15/33

What does a procedure describe?

- Capturing a common pattern

- (* 3 3)
- (* 25 25)
- (* foobar foobar)

(lambda (x) (* x x))

Common pattern to capture

Name for thing that changes

16/33

Modularity of common patterns

Here is a common pattern:

```
(sqrt (+ (* 3 3) (* 4 4)))
(sqrt (+ (* 9 9) (* 16 16)))
(sqrt (+ (* 4 4) (* 4 4)))
```

Here is one way to capture this pattern:

```
(define pythagoras
  (lambda (x y)
    (sqrt (+ (* x x) (* y y)))))
```

17/33

Modularity of common patterns

Here is a common pattern:

```
(sqrt (+ [red] [green]))
(sqrt (+ [red] [green]))
(sqrt (+ [red] [green]))
```

So here is a cleaner way of capturing the pattern:

```
(define square (lambda (x) (* x x)))
(define pythagoras
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))
```

18/33

Why?

- Breaking computation into modules that capture commonality
 - Enables reuse in other places (e.g. square)
- Isolates (abstracts away) details of computation within a procedure from use of the procedure
 - Useful even if used only *once* (i.e., a unique pattern)

```
(define (comp x y) (/ (+ (* x y) 17) (+ x y 4)))  
(define (comp x y) (/ (prod+17 x y) (sum+4 x y)))
```

19/33

Why?

- May be many ways to divide up

```
(define square (lambda (x) (* x x)))  
(define pythagoras  
  (lambda (x y)  
    (sqrt (+ (square x) (square y)))))  
  
(define square (lambda (x) (* x x)))  
(define sum-squares  
  (lambda (x y) (+ (square x) (square y))))  
(define pythagoras  
  (lambda (y x) (sqrt (sum-squares y x))))
```

20/33

Abstracting the process

- Stages in capturing common patterns of computation
 - Identify modules or stages of process
 - Capture each module within a procedural abstraction
 - Construct a procedure to control the interactions between the modules
 - Repeat the process within each module as necessary

21/33

A more complex example

- Remember our method for finding sqrts
 - To find the square root of X
 - Make a guess, called G
 - If G is close enough, stop
 - Else make a new guess by averaging G and X/G

The stages of “SQRT”

- When is something “close enough”
- How do we create a new guess
- How do we control the process of using the new guess in place of the old one

22/33

Procedural abstractions

For “close enough”:

```
(define close-enuf?  
  (lambda (guess x)  
    (< (abs (- (square guess) x)) 0.001)))
```



Note use of procedural abstraction!

23/33

Procedural abstractions

For “improve”:

```
(define average  
  (lambda (a b) (/ (+ a b) 2)))  
(define improve  
  (lambda (guess x)  
    (average guess (/ x guess))))
```

24/33

Why this modularity?

- “Average” is something we are likely to want in other computations, so only need to create once
- Abstraction lets us separate implementation details from use
 - Originally:

```
(define average
  (lambda (a b) (/ (+ a b) 2)))
```

- Could redefine as

```
(define average
  (lambda (x y) (* (+ x y) 0.5)))
```

- No other changes needed to procedures that use **average**
- Also note that variables (or parameters) are internal to procedure – cannot be referred to by name outside of scope of lambda

25/33

Controlling the process

- Basic idea:
 - Given X, G, want (**improve G X**) as new guess
 - Need to make a decision – for this need a new *special form*

```
(if <predicate> <consequence> <alternative>)
```

26/33

The IF special form

```
(if <predicate> <consequence> <alternative>)
```

- Evaluator first evaluates the **<predicate>** expression.
- If it evaluates to a TRUE value, then the evaluator evaluates and returns the value of the **<consequence>** expression.
- Otherwise, it evaluates and returns the value of the **<alternative>** expression.
- **Why must this be a special form? (i.e. why not just a regular lambda procedure?)**

27/33

Controlling the process

- Basic idea:
 - Given X, G, want (**improve G X**) as new guess
 - Need to make a decision – for this need a new *special form*
- So heart of process should be:

```
(if (close-enuf? G X)
    G
    (improve G X))
```

- But somehow we want to use the value returned by “improving” things as the new guess, and **repeat the process**

28/33

Controlling the process

- Basic idea:
 - Given X, G, want (**improve G X**) as new guess
 - Need to make a decision – for this need a new *special form*
 - So heart of process should be:
- ```
(define sqrt-loop (lambda (G X)
 (if (close-enuf? G X)
 G
 (sqrt-loop (improve G X) X))))
```
- But somehow we want to use the value returned by “improving” things as the new guess, and repeat the process
  - Call process **sqrt-loop** and reuse it!

29/33

## Putting it together

- Then we can create our procedure, by simply starting with some initial guess:

```
(define sqrt
 (lambda (x)
 (sqrt-loop 1.0 x)))
```

30/33

## Checking that it does the “right thing”

- Next lecture, we will see a formal way of tracing evolution of evaluation process
  - For now, just walk through basic steps
    - (`sqrt 2`)
      - (`sqrt-loop 1.0 2`)
      - (`if (close-enuf? 1.0 2) ...`)
      - (`sqrt-loop (improve 1.0 2) 2`)
- This is just like a normal combination
- (`sqrt-loop 1.5 2`)
  - (`if (close-enuf? 1.5 2) ...`)
  - (`sqrt-loop 1.4166666 2`)
- And so on...

31/33

## Abstracting the process

- Stages in capturing common patterns of computation
  - Identify modules or stages of process
  - Capture each module within a procedural abstraction
  - Construct a procedure to control the interactions between the modules
  - Repeat the process within each module as necessary

32/33

## Summarizing Scheme

- Primitives
  - Numbers **1, -2.5, 3.67e25**
  - Strings
  - Booleans

- Built in procedures  $*, +, -, /, =, >, <$


Means of Combination

- (procedure argument<sub>1</sub> argument<sub>2</sub> ... argument<sub>n</sub>)

- Means of Abstraction

- Lambda .  **Create a procedure**
- Define .  **Create names**

- Other forms

- if  $\cdot$   **Control order of evaluation**

- Creates a loop in system
  - allows abstraction of name for object

```
graph TD
 A[Create a procedure] --> B[Create a procedure]
```

### Create names

Control order of evaluation

33/33