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

A new idea: two worlds

visible world

expression

37

37

printed representation of value

execution world

100101

value

A new idea: two worlds

visible world

expression

37

37

printed representation of value

execution world

100101

value

names: look up value of name in current environment

Define special form

- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined

visible world

(define pi 3.14) "pi --> 3.14"

execution world

undefined

pi 3.14

name value

Mathematical operators are just names

(+ 3 5) ➞ 8
(define fred +) ➞ undef
(fred 4 6) ➞ 10

• How to explain this?

• Explanation
  - + is just a name
  - + is bound to a value which is a procedure
  - line 2 binds the name fred to that same value
Language elements -- procedures

- Need to capture ways of doing things -- use procedures

\[
\text{R u l e s  f o r } \text{evaluating} \\
1. \text{If self-evaluating, return value.} \\
2. \text{If a name, return value associated with name in environment.} \\
3. \text{If a special form, do something special.} \\
4. \text{If a combination, then} \\
   a. \text{Evaluate all of the subexpressions of combination (in any order)} \\
   b. \text{apply the operator to the values of the operands (arguments) and return result}
\]

- Rules for applying

A compound proc that squares its argument

Lambda: making new procedures

\[
\text{Rules for evaluating} \\
\text{Rules for applying}
\]

Rumplestiltskin effect!

\((\text{define square } \text{(lambda } (x) (* x x)))\)

Rumplestiltskin effect!

The power of naming things

Lambda: making new procedures

\[
\text{Rules for evaluating} \\
\text{Rules for applying}
\]

Scheme Basics

- 

A primitive proc that multiplies its arguments
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!

---

Lambda special form

- **lambda syntax**  
  `(lambda (x y) (/ (+ x y) 2))`
- 1st operand position: the parameter list  
  - a list of names (perhaps empty)  
  - determines the number of operands required
- 2nd operand position: the body  
  - 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:

---

The value of a lambda expression is a procedure…

*Om Mani Padme Hum…*

---

Using procedures to describe processes

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

---

What does a procedure describe?

- Capturing a common pattern  
  - `(* 3 3)`  
  - `(* 25 25)`  
  - `(* foobar foobar)`
- Common pattern to capture  
  - `(lambda (x) (* x x))`
- Name for thing that changes
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:

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

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)

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

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

A more complex example

- Remember our method for finding square roots
  - 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

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

Procedural abstractions

For “close enough”:

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

Note use of procedural abstraction!

Procedural abstractions

For “improve”:

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

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

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

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>)
```

The *IF* special form

\[
(\text{if} \ <\text{predicate}>\ <\text{consequence}>\ <\text{alternative}>)
\]

- 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?)

| 31/37 |

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
    \[
    (\text{if} \ <\text{predicate}>\ <\text{consequence}>\ <\text{alternative}>)
    \]
  - So heart of process should be:
    \[
    (\text{define} \ \text{sqrt-loop} \ (\lambda G X)
    \text{(if} \ (\text{close-enuf?} \ G \ X)
    \text{G}
    \text{(sqrt-loop} \ (\text{improve} \ G \ X) \ X )
    \]
  - 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!

| 32/37 |

Putting it together

- Then we can create our procedure, by simply starting with some initial guess:
  \[
  (\text{define} \ \text{sqrt}
  (\lambda (x)
  \text{(sqrt-loop} \ 1.0 \ x))
  \]

| 33/37 |

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

| 34/37 |

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...
Summarizing Scheme

- Primitives
  - Numbers: 1, -2.5, 3.67e25
  - Strings
  - Booleans
  - Built-in procedures: *, +, \( / \), =, >, <

- Means of Combination
  - (procedure argument1 argument2 ... argumentn)

- Means of Abstraction
  - Lambda
  - Define

- Other forms
  - if:

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

Create a procedure
Create names

Control order of evaluation