Basic Scheme
February 8, 2007

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

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

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

Read-Eval-Print

Visible world
(+ 3 (* 4 5))

EXECUTION WORLD
READ
Internal representation for expression
EVAL
Value of expression
PRINT
Visible world
23

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

This lecture
Adding procedures and procedural abstractions to capture processes
Language elements -- procedures

• Need to capture ways of doing things – use procedures

\[(\lambda (x) (* x x))\]

To process something multiply it by itself

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

\[\text{multiply it by itself}\]

Language elements -- procedures

• Use this anywhere you would use a procedure

\[\left((\lambda (x) (* x x))\right) 5\]

\[\text{multiply it by itself}\]

Language elements -- abstraction

• Use this anywhere you would use a procedure

\[\left((\lambda (x) (* x x))\right) 5\]

Don’t want to have to write obscurative code – so can give the lambda a name

\[\text{(define square (lambda (x) (* x x)) Rumplestiltskin effect! (The power of naming things)}}\]

\[\text{square 5} \Rightarrow 25\]

Scheme Basics

• Rules for evaluating

1. If self-evaluating, return value.
2. If a name, return value associated with name in environment.
3. If a special form, do something special.
4. If a combination, then
   a. Evaluate all of the subexpressions of combination (in any order)
   b. apply the operator to the values of the operands (arguments) and return result

• Rules for applying

1. If procedure is primitive procedure, just do it.
2. If procedure is a compound procedure, then evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

Interaction of define and lambda

1. \[\text{(lambda (x) (* x x))} \Rightarrow \#\{\text{compound-procedure 9}\}\]
2. \[\text{(define square (lambda (x) (* x x))} \Rightarrow \text{undef}\]
3. \[\text{square 4} \Rightarrow 16\]
4. \[\left((\lambda (x) (* x x))\right) 4 \Rightarrow 16\]
5. \[\text{(define (square x) (* x x))} \Rightarrow \text{undef}\]

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

Lambda special form

• Lambda syntax \[\left(\lambda (x\ y) \left(\left(\text{begin} x\ y\ \right)\right)\right)\]

• 1st operand position: the parameter list \[\left(\left(\text{begin} x\ y\ \right)\right)\]
  – a list of names (perhaps empty) \[\left(\ \right)\]
  – determines the number of operands required

• 2nd operand position: the body \[\left(\left(\text{begin} x\ y\ \right)\right)\]
  – 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: \[\text{(define x (lambda (y) (+ 3 2)))}\]
  • \[\text{x}\]
  • \[\text{y}\]
  • \[\text{semantics of lambda}\


THE VALUE OF A LAMBDA EXPRESSION IS A PROCEDURE

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)

Here is a common pattern:

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

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)))))
```

Modularity of common patterns

Here is a common pattern:

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

Here is one way to capture this pattern:

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

Modularity of common patterns

Here is a common pattern:

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

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))))))
```
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)))
```

Why?

- May be many ways to divide up

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

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 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

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:
    ```scheme```
    ```
    (define average
      (lambda (a b) (/ (+ a b) 2)))
    ```
  - Could redefine as
    ```scheme```
    ```
    (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
    ```scheme```
    ```
    (if <predicate> <consequence> <alternative>)
    ```

The `IF` special form

```scheme```
```(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?)

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
    ```scheme```
    ```
    (if <predicate> <consequence> <alternative>)
    ```
  - So heart of process should be:
    ```scheme```
    ```
    (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!

Putting it together

- Then we can create our procedure, by simply starting with some initial guess:
  ```scheme```
  ```
  (define sqrt
    (lambda (x)
      (sqrt-loop 1.0 x)))
  ```
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

  • \((\text{sqrt } 2)\)
    • \((\text{sqrt-loop } 1.0 \ 2)\)
    • \((\text{if } (\text{close-enuf? } 1.0 \ 2) \ldots )\)
    • \((\text{sqrt-loop } (\text{improve } 1.0 \ 2) \ 2)\)
  
  This is just like a normal combination
  • \((\text{sqrt-loop } 1.5 \ 2)\)
  • \((\text{if } (\text{close-enuf? } 1.5 \ 2) \ldots )\)
  • \((\text{sqrt-loop } 1.41666666 \ 2)\)
  
• And so on…

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

Summarizing Scheme

• Primitives
  – Numbers 1, -2.5, 3.67e25
  – Strings
  – Booleans
  – Built in procedures * , +, -, /, =, >, <, ...\n
• Means of Combination
  – \(\text{(procedure argument } 1 \ \text{argument2 } \ldots \ \text{argument n})\)

• Means of Abstraction
  – Lambda
  – Define

• Other forms
  – if

Create a procedure, allows abstraction of name for object

Create names

Create a loop in system, allows control of evaluation

Control order of evaluation