6.001 Recitation 2: More Scheme

RI: Gerald Dalley
9 Feb 2007

Announcements / Notes

- Lecture 2, slide 29 has 4 missing parentheses.
- Sugarless lambda – (a) keeps it clear that creating a procedure and assigning it to a name are two distinct steps, and (b) often we don't need a name – we'll see many examples of this later.
- ```
  (if (is-serious?
        (scheduling-problem? you))
      (email dkauf@mit.edu)
      (attend-section-we-assigned you))
  ```
- First tutorial is **Monday** or **Tuesday**
- Mid-semester recitation feedback
- InstaQuiz returned
- DrScheme options: case sensitivity & rationality

The `lambda` Special Form

`(lambda` *parameters body*`)`

Creates a procedure with the given parameters and body. *parameters* is a list of names of variables. *body* is one or more scheme expressions. When the procedure is applied, the body expressions are evaluated in order and the value of the last one is returned.

Evaluate the following expressions:

```
(lambda (x) x)
==> a procedure
```

```
((lambda (x) x) 17)
==> 17
```

```
((lambda (x y) x) 42 17)
==> 42
```

```
((lambda (x y) y) (/ 1 0) 3)
==> error
```

```
((lambda (x y) (x y 3))
 (lambda (a b) (+ a b)) 14)
--> ((lambda (a b) (+ a b)) 14 3)
--> (+ 14 3)
==> 17
```

The `if` Special Form

`(if` *test consequent alternative*`)`

If the value of the test is not false (`#f`), evaluate the consequent, otherwise evaluate the alternative.

Why must this be a special form?

If we used a `lambda`, both the consequent and alternative would always be evaluated. This would result in infinite loops for recursive procedures.

Does `if` *give us new functionality?*

Yes, see the previous answer

Evaluate the following expressions (assuming `x` *is bound to* 3*):*

```
(if #t (+ 1 1) 17)
==> 2
```

```
(if #f #f 42)
==> 42
```

```
(if (> x 0) x (- x))
==> 3
```

```
(if 0 1 2)
==> 1
```

```
(if x 7 (7))
==> 7
```

Write the body of the following procedure:

```
;; If x is not the same as the expected
;; value, some illegal expression is
;; evaluated.
;;
;; Hints: (equals? x y) can be used to test
;; for equivalence and (not x) flips true/
;; false values.
(define (check x expected)
  (if (not (equal? x expected))
      ("error")))
```
```

The `cond` Special Form

```
(cond (test-expr1 expr ...)
      (test-expr2 expr ...)
      (else expr ...))
```

Evaluation rules:

1. Evaluate *test-expr1*
2. If the value is not false (`#f`), evaluate the rest of the associated expressions and return the last value.
3. Otherwise, continue to the next test expression and repeat.
4. If no test expressions are non-false, evaluate the `else` clause and return the value of the last expression, if an `else` clause exists.

Why must this be a special form?

For the same reasons `if` cannot be implemented with just `lambda`.

Does `cond` *give us new functionality?*

No, it's just a sugary way of doing complex `if` expressions.

Evaluate the following expressions (assuming `x` *is bound to* 3*):*

```
(cond ((= 1 x) "one")
      ((= 2 x) "two")
      ((= 3 x) "three"))
==> 3
```

```
(cond (((lambda (x) (= 3 x)) x) "three")
      (else                     "not␣three"))
==> "three"
```

```
(cond ((lambda (x) (= 2 x)) "two")
      (else                 "not␣two"))
==> "two"
```

Biggie Size!

Suppose we're designing an point-of-sale and order-tracking system for Wendy's[1]. Luckily the Über-Qwuick drive through supports only 4 options: Classic Single Combo (hamburger with one patty), Classic Double With Cheese Combo (2 patties), and Classic Triple with Cheese Combo (3 patties), Avant-Garde Quadruple with Guacamole Combo (4 patties). We shall encode these combos as 1, 2, 3, and 4 respectively. Each meal can be *biggie-sized* to acquire a larger box of fries and drink. A *biggie-sized* combo is represented by 5, 6, 7, and 8 respectively.

1. Write a procedure named `biggie-size` which when given a regular combo returns a *biggie-sized* version.

```
(define biggie-size
       (lambda (combo)
               (+ combo 4)))
;;tests
(check (biggie-size 1) 5) ; single -> biggie single
(check (biggie-size 4) 8) ; avant  -> biggie avant
```

2. Write a procedure named `unbiggie-size` which when given a *biggie-sized* combo returns a non-*biggie-sized* version.

```
(define unbiggie-size
        (lambda (combo)
                (- combo 4)))
;;tests
(check (unbiggie-size 5) 1) ; single <- biggie single
(check (unbiggie-size 8) 4) ; avant  <- biggie avant
```

3. Write a procedure named `biggie-size?` which when given a combo, returns true if the combo has been *biggie-sized* and false otherwise.

```
(define biggie-size?
        (lambda (combo)
                (> combo 4)))
;;tests
(check (biggie-size? 1) #f)
(check (biggie-size? 7) #t)
```

4. Write a procedure named `combo-price` which takes a combo and returns the price of the combo. Each patty costs $1.17, and a *biggie-sized* version costs $.50 extra overall.

```
(define combo-price
        (lambda (combo)
                (if (biggie-size? combo)
                    (+ (combo-price (unbiggie-size combo)) 0.50)
                    (* 1.17 combo))))
;; tests
(check (combo-price 1) 1.17)
(check (combo-price 2) 2.34)
(check (combo-price 5) 1.67)
```

5. An order is a collection of combos. We'll encode an order as each digit representing a combo. For example, the order 237 represents a Double, Triple, and *biggie-sized* Triple. Write a procedure named `empty-order` which takes no arguments and returns an empty order.

```
(define empty-order
        (lambda () 0))
;; no tests are practical yet
```

6. Write a procedure named `add-to-order` which takes an order and a combo and returns a new order which contains the contents of the old order and the new combo. For example, `(add-to-order 1 2) -> 12`.

```
(define add-to-order
        (lambda (order combo)
                (+ (* order 10) combo)))
;; tests
(check (add-to-order 1 2) 12)
```

7. *Write a procedure named `order-size` which takes an order and returns the number of combos in the order. For example, `(order-size 237) -> 3`. You may find `quotient` (integer division) useful.

```
(define order-size
  (lambda (order)
    (if (= 0 order)
        0
        (+ 1 (order-size (quotient order 10))))))
;; tests
(check (order-size (empty-order)) 0)
(check (order-size 237) 3)
(check (order-size (add-to-order (empty-order) 2)) 1)
(check (order-size (add-to-order (biggie-size 3) 2)) 2)
```

8. *Write a procedure named `order-cost` which takes an order and returns the total cost of all the combos. In addition to `quotient`, you may find `remainder` (computes remainder of division) useful.

```
(define order-cost
  (lambda (order)
    (if (= 0 order)
        0
        (+ (combo-price (remainder order 10))
           (order-cost (quotient order 10))))))
;; tests
(check (order-cost (empty-order)) 0)
(check (order-cost 1) 1.17)
(check (order-cost 12) (* 1.17 3))
(check (order-cost 58) (+ (* 1.17 5) (* 2 0.50)))
```