CS151.02 2010S *Functional Problem Solving*

Class 53: Objects in Scheme

Held: Monday, May 10, 2010

Summary: We begin to explore *objects* in Scheme. Objects encapsulate data and the procedures that work with those data.

Related Pages:

- EBoard.
- Lab: Building Objects in Scheme.
- Reading: Building Objects in Scheme.

Notes:

- EC for Student Art Salon.
- I intend to go over exam 3 in class tomorrow. I expect you all to be there.
- Special EC for filling out the end-of-semester RISC survey at http://www.grinnell.edu/academic/psychology/faculty/dl/risc/.

Overview:

- Motivating problems: Circles, turtles, and counters.
- Building and using compound values.
- Objects: A new approach to compound values.
- Creating objects in Scheme.

Motivating Problems

- Suppose we want to make a drawing composed of a lot of circles. How should we represent the circles?
- How might we implement a counter to keep track of the number of times we do procedure calls?
- How might we implement turtles?

Compound Values

- We've seen two ways to represent such compound values.
  - As a list
  - As a vector
- Once we've chosen a representation, we then build a set of procedures that work with that representation.
- As we've seen in our experiments with representing compound values, there are strengths and weaknesses to simply choosing a representation and writing procedures to work with that

implementation.

- Strengths: Access parts by procedure; Relatively easy to use.
- Weaknesses: Not fully encapsulated; hard to separate core operations from external operations (since they're called the same way); hard to limit access.

Objects: Encapsulating Values and Operations

- In the late 1960's, some computer scientists decided to extend the idea of representing data into something they call an *object*
- Objects group data.
- Objects also do things.
- You can't directly access the parts of an object.
- Rather, you ask the object to do things or tell you things using their internal state.
- The requests you send to objects are called *messages*.
- Traditional objects also provide a number of other advantages. Nonetheless, we'll focus on encapsulation.

Objects in Scheme

- Standard Scheme doesn't include objects as a built-in type. Hence, we have to implement them ourselves.
    - But there's a common technique that Scheme programmers use.
- The trick that we recommend is that you implement objects as procedures that take a *message* as a parameter.
- Traditionally, the messages begin with a colon.
- Here's a simple object that will respond when you greet it or leave it.

```
(define greeter
  (lambda (message)
    (cond
      ((eq? message ':enter) (display "Hello") (newline))
      ((eq? message ':leave) (display "Goodbye") (newline))
      (else (error "Unknown Message")))))
```

- Here's how we use it

```
> (greeter ':enter)
Hello
> (greeter ':leave)
Goodbye
> (greeter ':sleep)
Unknown Message
```

Adding State

- But how do we have an object keep track of information about itself?
- We build a local symbol table that is only accessible to the procedure.
- We can build such a table by putting a `let` *outside* the `lambda` for the procedure.

```
(define fixed-value
  (let ((value 5))
    (lambda (message)
      (cond
        ((eq? message ':get) value)
        (else (error "fixed-value:" "unknown message"))))))
```

- Typically, we use vectors to encapsulate our state because we know how to mutate vectors.

```
(define incrementable-value
  (let ((value (vector 0)))
    (lambda (message)
      (cond
        ((eq? message ':get) (vector-ref value 0))
        ((eq? message ':add1!)
         (vector-set! value 0
                      (+ 1 (vector-ref value 0))))
        (else (error "fixed-value:" "unknown message"))))))
```

- And an example of its use

```
> (incrementable-value ':get)
0
> (incrementable-value ':add1!)
> (incrementable-value ':get)
1
```

Lab

- If there is time (unlikely), you can begin the lab on object-oriented programming.
