COMP 210, FALL 2000
Lecture 34: The Search for Identity

Reminders

1. The last homework is due today, in class.
2. The last exam is available today, in class. If you missed class, check the box outside my office, DH 2065.
3. Review session will be held Monday, during class period. Wednesday, I'll talk about how computers really work (as opposed to the high-level view we've had of how programming languages work). Friday, we'll fill out teacher evaluations.

Equality

To introduce this subject neatly, it needs to segue from a lecture about some program that used structures. So, I'm going to back up, mentally, to Monday's lecture. Recall in Monday's lecture that we built an address book to show set-structure! The address book included a structure, called an entry, that looked something like

```
;; an entry is a structure
;;   (make-entry name phone)
;; where name is a symbol and phone is a number
(define-struct  entry  (name phone))
```

At some point, in working our way through the program update, we performed the action

```
(set-entry-phone! (first a-book) '7133486013)
```

and this was purported to change the contents of the particular entry that was at the head of the list denoted by a-book.

This notion–-that the contents of a variable or a structure are mutable–-leads to a deep question (that some of us find a fascinating question): when are two structures or two lists the same? With variables, we can use the Scheme operator =. With symbols, we can use the Scheme operator symbol=? Can we ask the same kind of question for structures or lists? (Is there a "list=?" and what would it mean?)

What if we type

```
(make-entry 'keith  7133486013)
(make-entry 'keith  7133486013)
```

Are the two structures that this creates the same? What does that question mean? There are two possible answers for an entry.

1. Are the structures equivalent? That is, do they have the same values and the same behavior?
2. Are the two structures implemented with the same object in DrScheme?

As you could guess (from the fact that I am lecturing on the subject), Scheme provides a mechanism to answer each of these questions.

The first operator, called equal?, tests whether two arbitrary values have the same internal structure and the same value in every position in that structure. Thus,

```
(equal?  4 4)
> true
(equal?  ‘a ‘a)
> true
(equal? ‘a ‘A)
> false
(equal? 4  ‘a)
> false
```

Remember that = and symbol=? had pretty strict contracts.

= : number number → boolean

symbol=?: symbol symbol → boolean

The contract for equal? is more general than either of these.

equal? : alpha beta → boolean

If we try equal? on a more complex example, it does the intuitive thing:

```
(equal?  (make-entry ‘keith 7133486013) (make-entry ‘keith 7133486013))
> true
(equal?  (list 1 2 3) (list 1 2 3))
> true
(equal?  (list 1 2 3) (list 1 2 4))
> false
```

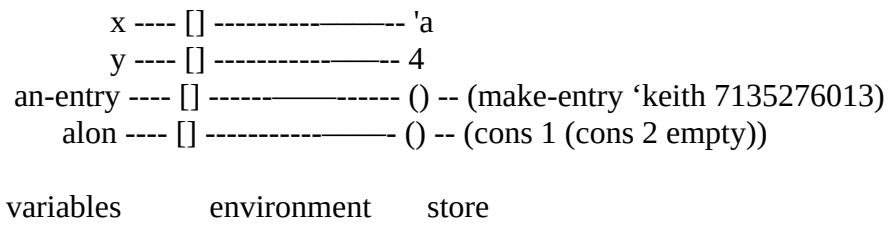
The operator equal? tests for identical form and identical values.

But are ‘a and ‘a the same object? What about (list 1 2 3) and (list 1 2 3)? Scheme provides another mechanism to allow you to test whether or not two names are implemented by the same object. The operator eq? does this. You can use eq? as a tool to understand more about how Scheme actually stores objects in its memory.

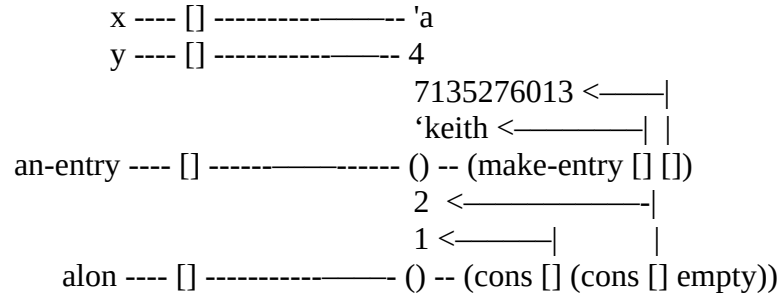
```
(eq?  (list 1 2 3)  (list 1 2 3)
> false
(define a (list 1 2 3))
(define b a)
(eq? a b)
> true
(define c (list 1 2 3))
(equal? b c)
> true
(eq? b c)
> false
(set! b c)
(equal? b c)
> true
(eq? b c)
```

So far, this relies on intuition. Can we make it more concrete? (Yes. Take 311)

To understand this at a level beyond the intuitive requires an understanding of how Scheme organizes its memory. In the study of programming languages (and, after all, you have begun the study of programming languages), we call this the “memory model” of a programming language.Scheme's memory model contains two main layers, denoted by [] and () below:



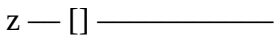
The boxes are references to a piece of data. Thus, x is mapped to a particular box (reference) which refers to ‘a. Similarly, y is mapped to a box that refers to 4. Scheme is uniform in its treatment of data, so the way that we’ve represented the list and the structure are misleading. They contain boxes that refer to values.



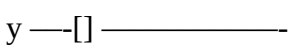
So, Scheme creates a mapping from names that you can write in a program to specific boxes (or references). These are the lines from a variable name to a box. It also creates and maintains a mapping from specific boxes to specific values.

To really understand define, set!, set-structure!, you need to see how they affect the memory model. With this understanding, the distinction between equal? and eq? becomes clear.

- define adds a name to the mapping and makes its box refer to some specific value. Thus, (define z 17) simply adds a new line to the memory model.

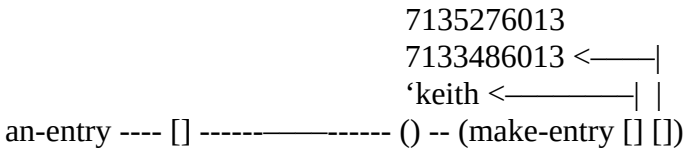
  

- set! changes the mapping from a name to a box. Thus, (set! y 7) would create a different line for y

  while leaving behind the value for 4. Does the box for 4 remain? That’s a good question. If any other name is mapped to 4, then it remains. This includes an implicit name inside some structure.

- set-structure! Changes the value inside the structure. It replaces the box inside the structure with another box, but leaves the mapping of names to boxes intact. Thus, (set-entry-phone! An-entry 7133486013) creates the memory model



Some values are represented uniquely. For example, numbers always have the same name (4 is 4), so any use of 4 refers to the same box. Similarly with symbols.

With this new understanding, we can explain eq? It just tests to see whether or not two variables refer to the same box. On the other hand, equal? is a recursive function that tests, at each level, whether or not the layout is the same, and the values are the same.