

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 1999

**Recitation – Friday, March 12**

## 1. Equality Predicates

In the past, we have used the function `eq?` to test equality between two symbols and `=` to test equality between two numbers. The built-in function `eqv?` tests the equality of any two atoms.

|  |  |   |
|--|--|---|
| <code>(= 2 2)</code>                         | <code>(= 'robot 'robot)</code>           | <code>(= (cons 2 4) (cons 2 4))</code>    |
| $\Rightarrow$ <code>#t</code>                | $\Rightarrow$ <code>&lt;error&gt;</code> | $\Rightarrow$ <code>&lt;error&gt;</code>  |
| <code>(eq? 2 2)</code>                       | <code>(eq? 'robot 'robot)</code>         | <code>(eq? (cons 2 4) (cons 2 4))</code>  |
| $\Rightarrow$ <code>&lt;undefined&gt;</code> | $\Rightarrow$ <code>#t</code>            | $\Rightarrow$ <code>#f</code>             |
| <code>(eqv? 2 2)</code>                      | <code>(eqv? 'robot 'robot)</code>        | <code>(eqv? (cons 2 4) (cons 2 4))</code> |
| $\Rightarrow$ <code>#t</code>                | $\Rightarrow$ <code>#t</code>            | $\Rightarrow$ <code>#f</code>             |

That's annoying. What if we want to check to see if two lists or tree structures are the same? There's a function called `equal?` that does this. Here are some examples. How could we write `equal?`?

|                                  |                               |   |                               |
|----------------------------------|-------------------------------|---|-------------------------------|
| <code>(equal? 5 5)</code>        | $\Rightarrow$ <code>#t</code> | <code>(equal? '(this is list 1) '(this is list 1))</code>   | $\Rightarrow$ <code>#t</code> |
| <code>(equal? 5 6)</code>        | $\Rightarrow$ <code>#f</code> | <code>(equal? '(this is list 1) '(this is (list 1)))</code> | $\Rightarrow$ <code>#f</code> |
| <code>(equal? 'tree 'cat)</code> | $\Rightarrow$ <code>#f</code> | <code>(equal? '(lambda (x) x) '(lambda (x) x))</code>       | $\Rightarrow$ <code>#t</code> |
| <code>(equal? 'cat 'cat)</code>  | $\Rightarrow$ <code>#t</code> | <code>(equal? (lambda (x) x) (lambda (x) x))</code>         | $\Rightarrow$ <code>#f</code> |

`(define (equal? x y)`

)

## 2. Quote, Quasiquote, Unquote

Here are examples using Quote (`'`) Quasiquote (```) and Unquote (`,`). Quasiquote is the same as quote, except it cancels out with unquote. Consider the following examples: `(define y 5)`

|                              |  |
|------------------------------|--|
| <code>'y</code>              | $\Rightarrow$ <code>y</code>                     |
| <code>'(x y z)</code>        | $\Rightarrow$ <code>(list 'x 'y 'z)</code>       |
| <code>'y</code>              | $\Rightarrow$ <code>y</code>                     |
| <code>'5</code>              | $\Rightarrow$ <code>5</code>                     |
| <code>'(x y z)</code>        | $\Rightarrow$ <code>(list 'x 'y 'z)</code>       |
| <code>'(x ,y z)</code>       | $\Rightarrow$ <code>(list 'x 'y 'z)</code>       |
| <code>'(x ,y z)</code>       | $\Rightarrow$ <code>(list 'x 5 'z)</code>        |
| <code>'(x (+ 2 5) z)</code>  | $\Rightarrow$ <code>(list 'x '(+ 2 5) 'z)</code> |
| <code>'(x ,(+ 2 5) z)</code> | $\Rightarrow$ <code>(list 'x 7 'z)</code>        |

## 3. Continuations - Where do you want to go next?

A couple of weeks ago, we saw an example of why we couldn't write our own `if` statement, because `if` needs to be a special form. Recall the following:

```
(define (c-if pred consequent alt)
  (if (zero? pred) alt consequent))
(c-if (= 1 1) 5 (/ 1 0))  $\Rightarrow$  <error> [even though you might think it should return 5]
```

But we could write a procedure like this. Look carefully at what this is doing.

```
(define (my-if pred consequent-proc alt-proc)
  (if (zero? pred) (alt-proc) (consequent-proc)))
```

How would you rewrite `(c-if (= 1 1) 5 (/ 1 0))` using `my-if` so that it does the right thing (returns 5)?

## 4. Replace

We are going to build a function called `replace` which searches through a tree and replaces items according to certain matching rules. A call to `replace` looks like this: `(replace match? pattern change tree)`, where `match?` is a predicate called on the elements of the tree, `pattern` is the pattern to be matched, `change` is a function that changes the part of the tree that matched, and `tree` is the tree to be operated upon. We can use this function to write procedures that replace one symbol (`'hate`) with another (`'love`) or to find the absolute value of the leaves of a tree:

```
(define (censor tree)
  (replace eq? 'hate (lambda (x) 'love) tree))
(censor '(I hate scheme))
⇒ (I love scheme)
(censor '(I ((hate) scheme)))
⇒ (I ((love) scheme))

(define (abs-tree tree)
  (replace (lambda (x y) (and (number? x) (< x y)))
    0 - tree))
(abs-tree '(-3 -2 (-1 0 1) 2 3))
⇒ (3 2 (1 0 1) 2 3)
```

Fill in the code for the function `replace`:

```
(define (replace match? pat change tree)
```

## 5. Requal?

Now we want to allow patterns that have a little more flexibility by introducing wildcards. A wildcard is a special symbol which matches a more general class of patterns. We will start by using the symbol `'?` to match any one element of a list. To do this, we write a function called `requal?` that takes a list `x` and a pattern `pat` and returns `#t` if the pattern matches list. For example,

```
(requal? '(1 2 3 4) '(1 2 3 ?))
⇒ #t
(requal? '(1 2 3 4) '(1 ? 3 ?))
⇒ #t

(requal? '(1 (2 3) 4) '(1 ? 4))
⇒ #t
(requal? '(1 2 3 4) '(1 ? 4))
⇒ #f
```

Now write the code for the function `requal?`:

```
(define (requal? x pat)
```

)

## 5. Pattern Matching

Fill in the correct values of `pred`, `pattern`, and `change` to make these expressions work:

```
(replace pred pattern change
  '(M (M V T) (M (X Y) T) B))
⇒ (M (M I T) (M I T) B)
pred =
pattern =
change =

(replace pred pattern change
  '(1 (2 3) (negate (4 5 6)) 7 (negate (8 9))))
⇒ (1 (2 3) (-4 -5 -6) 7 (-8 -9))
pred =
pattern =
change =
```

The function `replace` performs one of the recursive paths that were demonstrated in lecture yesterday. You can think about the pattern matcher from lecture as a higher-level function that performs two more levels of recursion on our existing `replace` function. (1) Whenever `replace` is called, it operates not just on one replacement rule, but on a set of replacement rules. (2) The higher-level function recursively calls `replace` on the result of `replace` until nothing changes.