MASSACHVSETTS INSTITVTE 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?` is a tests the equality of any two atoms.

```
(= 2 2)              (= 'robot 'robot)         (= (cons 2 4) (cons 2 4))
⇒ #t                 ⇒ <error>                 ⇒ <error>
(eq? 2 2)            (eq? 'robot 'robot)       (eq? (cons 2 4) (cons 2 4))
⇒ <undefined>        ⇒ #t                      ⇒ #f
(eqv? 2 2)           (eqv? 'robot 'robot)      (eqv? (cons 2 4) (cons 2 4))
⇒ #t                 ⇒ #t                      ⇒ #f
```

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?` ?

```
(equal? 5 5)         ⇒ #t      (equal? '(this is list 1) '(this is list 1))      ⇒ #t
(equal? 5 6)         ⇒ #f      (equal? '(this is list 1) '(this is (list 1)))    ⇒ #f
(equal? 'tree 'cat)  ⇒ #f      (equal? '(lambda (x) x) '(lambda (x) x))          ⇒ #t
(equal? 'cat 'cat)   ⇒ #t      (equal? (lambda (x) x) (lambda (x) x))            ⇒ #f
```

```
(define (equal? x y)
  (cond ((and (atom? x) (atom? y)) (eqv? x y))
        ((and (pair? x) (pair? y))
         (and (equal? (car x) (car y))
              (equal? (cdr x) (cdr y))))
        (else #f)) )
```

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)`

```
'y               ⇒ y
'(x y z)         ⇒ (list 'x 'y 'z)
`y               ⇒ y
`,y              ⇒ 5
`(x y z)         ⇒ (list `x `y `z)
`(x ,y z)        ⇒ (list `x `,y `z)        ⇒ (list `x 5 `z)
`(x (+ 2 5) z)   ⇒ (list `x `(+ 2 5) `z)   ⇒ (list `x (list `+ `2 `5) `z)
`(x ,(+ 2 5) z)  ⇒ (list `x `,(+ 2 5) `z)  ⇒ (list `x 7 `z)
```

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 conseq alt)
  (if (zero? pred) alt conseq))
(c-if (= 1 1) 5 (/ 1 0)) ⇒ <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 conseq-proc alt-proc)
  (if (zero? pred) (alt-proc) (conseq-proc)))
```

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

```
(my-if (= 1 1) (lambda () 5) (lambda () (/ 1 0)))
```

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)
  (cond ((null? tree) nil)
        ((match? tree pat) (change tree))
        ((not (pair? tree)) tree)
        (else (cons (replace match? pat change (car tree))
                    (replace match? pat change (cdr 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)
  (cond ((eq? pat '?) #t)
        ((and (atom? x) (atom? pat)) (eqv? x pat))
        ((and (pair? x) (pair? pat))
          (and (requal? (car x) (car pat))
               (requal? (cdr x) (cdr pat))))
        (else #f)) )
```

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    = requal?
pattern = '(M ? T)
change  = (lambda (x) '(M I T))
```

```
(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    = requal?
pattern = '(negate ?)
change  = (lambda (x) (map - (cadr x)))
```

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.