another look at the cmp function

brief

We have written a function to compare 2 lists & later expanded it so the first list can take a wild-card character which (for the purposes of our comparisons) will be assumed to match any item in the second list.

We used "=" as our wild-card symbol so...

```
(cmp '(bat = rat) '(bat cat rat))  →  t
(cmp '(bat = rat) '(bat (cat mat) rat))  →  t

; NB: cannot use wild-cards in 2nd arg so...
(cmp '(bat cat rat) '(bat = rat))  →  nil
```

our first version

used a typical structure which operated at the level of lists (ie: it did not descend to the next level of out structure – to the level of atoms).

Ignoring the use of wild-cards (so writing our own version of the "equal" function) we built the following...

```
(defun cmp (pat lis)
  (cond ((and (null pat) (null lis))
         t
         )
        ((or (null pat) (null lis))
         nil
         )
        ((eq (first pat) (first lis))
         (cmp (rest pat) (rest lis))
         )
        ((and (listp (first pat))
              (listp (first lis)))
         (and (cmp (first pat) (first lis))
              (cmp (rest pat) (rest lis)))
         )
        (t
         nil
         )
        ))
```

a new approach

this function starts by providing a way of comparing symbols (using eq) then works on nested lists (we will call them "trees" soon) by descending through list structures to the level of symbols...

```
(defun cmp (p l)
  (cond ((eq p l)                          ;; p = l so return true
         t
         )
        ((and (consp p) (consp l))         ;; p & l both cons-pairs
         (and (cmp (first p) (first l))    ;; so check firsts match...
              (cmp (rest p)(rest l)))      ;; ...and rests match
         )
        (t nil)                            ;; works without this clause...
        ))                                 ;; ...but better style
```

a few tests...

```
cg-user(12): (cmp nil nil)
t
cg-user(13): (cmp 'peach 'peach)
t
cg-user(14): (cmp 'peach 'pear)
nil
cg-user(15): (cmp '(a b c) '(a b c))
t
cg-user(16): (cmp '(a b c) '(a x c))
nil
cg-user(17): (cmp '(a b (p (x y z) nil q) c d)
                  '(a b (p (x y z) nil q) c d))
t
cg-user(18): (cmp '(a b (p (x y z) nil q) c d)
                  '(a b (p (x y z) zzz q) c d))
nil
```

better testing?

We will think about how to build test harnesses soon-ish but for now we can simply wrap tests in a list. Note we ensure all tests will return **t** by using **not** on those that return nil...

```
(defun test-all ()
  (list
   (cmp nil nil)
   (not (cmp nil 'banana))
   (not (cmp 'banana nil))
   (cmp 'peach 'peach)
   (not (cmp 'peach 'pear))
   (cmp '(a b c) '(a b c))
   (not (cmp '(a b c) '(a x c)))
   (cmp '(a b (p (x y z) nil q) c d)
        '(a b (p (x y z) nil q) c d))
   (not (cmp '(a b (p (x y z) nil q) c d)
             '(a b (p (x y z) zzz q) c d)))
   (cmp '(a b (p (x y . z) nil q) c d)
        '(a b (p (x y . z) nil q) c d))
   ))

cg-user(19): (test-all)
(t t t t t t t t t)
```

adding wild-cards

the functionality for handling wild-cards can now be added at the symbol level…

```
(defun cmp (p l)
  (cond ((eq p l)
         t
         )
        ((eq p '=)
         t
         )
        ((and (consp p) (consp l))
         (and (cmp (first p) (first l))
              (cmp (rest p)(rest l)))
         )
        (t nil)
        ))
```

NB: as with all of these functions you should test them yourself. The challenge is: can you find legitimate test cases where they give unexpected/incorrect results?

testing...

```
;; an updated set of tests
(defun test-all2 ()
  (list
   (cmp nil nil)
   (not (cmp nil 'banana))
   (not (cmp 'banana nil))
   (cmp 'peach 'peach)
   (not (cmp 'peach 'pear))
   (cmp '= 'banana)
   (cmp '= '(x y z))
   (cmp '(a b c) '(a b c))
   (not (cmp '(a b c) '(a x c)))
   (cmp '(a = c) '(a x c))
   (cmp '(a = =) '(a x c))
   (cmp '(=) '((a x c)))
   (cmp '(a b (p (x y z) nil q) c d)
        '(a b (p (x y z) nil q) c d))
   (not (cmp '(a b (p (x y z) nil q) c d)
             '(a b (p (x y z) zzz q) c d)))
   (cmp '(a = (p = nil q) = d)
        '(a b (p (x y z) nil q) c d))
   (cmp '(a b (p (x y . z) nil q) c d)
        '(a b (p (x y . z) nil q) c d))
   ))


;; use pprint so all the output is shown
cg-user(20): (pprint (test-all2))
(t t t t t t t t t t t t t t t)
```

a rewrite

it is interesting to read through the last solution we produced. One way to consider the function is: as a predicate which returns true iff (i) its arguments are equal or (ii) its first argument is a wild-card or (iii) both args are conses whose firsts are equal and whose rests are also equal.

Based on this phrasing we can rewrite the function again...

```
(defun cmp (p l)
  (or (eq p l)
      (eq p '=)
      (and (consp p)
           (consp l)
           (cmp (first p) (first l))
           (cmp (rest p)(rest l)))
      ))
```

It is often a matter of taste but try to understand the difference in focus that derives the function based on cond and the second using or & and.