Lisp Section Examples

Jerry's section handout from 5 years ago when he TA'ed CS107.

Problem 1

LISP has an immense library of functions. Lots of them can be implemented with basic functions; people just wrote standard versions and put them into libraries because these basic routines are commonly used. Let's assume that the version of LISP that you have doesn't include the append function. Write your own append. It should take two lists as its arguments and return a new list that contains all the elements of the first list added onto the second list.

Solution

```
;;;;;;;;;
;;; Function: my-append
;;;;;;;;;
;;; append takes two lists as its arguments.  It returns a list
;;; containing the elements of both lists in the following order:
;;; all the members of the first list, followed by all the members of the
;;; second
;;;;;;;;;

(defun my-append (list1 list2)
  (if (null list1)
      list2
      (cons (car list1) (my-append (cdr list1) list2))
  ))
```

I'm compelled to mention three things about the solution that have nothing to do with the functionality of the solution itself. First of all, you must comment your code! Commenting is nice for the reader in C, but if anything it's more vital with LISP. Many people see a sea of parentheses and don't think in prefix notation, so reading LISP code that hasn't been well-commented can be frustrating.

Second, it is just as important to choose terse yet descriptive names for your function names and variables. Third, good formatting is vital when the only real syntax is composed of nothing but parentheses. Even though the code is the same, there is an *extreme* disparity between the readability of the previous answer with the following code, even though they do the same thing!

```
(defun app (l1 l2)
  (if (null l1) l2 (cons (car l1) (app (cdr l1) l2))))
```

When you're dealing with a function more complex than append, this makes a really big difference.

Problem 2

Here's a problem that's similar to append in some ways: write a procedure that *interleaves* two lists. It should take two lists as arguments and return a list containing the elements of both
lists, alternating between the elements from the first list and elements from the second.

Example: > (interleave '(a b (c d) e f) '(1 2 (3 4) 5 6))
(a 1 b 2 (c d) (3 4) e 5 f 6)

Solution 1

```
;;;;;;;;;
;;; Function: interleave
;;;;;;;;;
;;; odds takes two lists as arguments, and produces a list containing the
;;; elements of both.  The elements will be returned in alternating order
;;; starting the with the 1st element of the first list, followed by the
;;; 1st element of the second list, followed by the 2nd element of the first
;;; list, and so on...
;;; When one of the two argument lists is longer than the other, the elements
;;; alternate until one of the lists runs out of elements, followed by all
;;; the remaining elements of the longer list.
;;;;;;;;;

(defun interleave (list1 list2)
  (cond  ( (null list1)   list2)
         ( (null list2)   list1)
         ( T  (cons (car list1)
                    (cons (car list2) (interleave (cdr list1) (cdr list2)))
              )
         )))
```

Solution 2

```
;;;;;;;;;
;;; Function:  interleave
;;;;;;;;;
;;; This performs the same function, but the code is written slightly differently.
;;; Instead of taking the first of both lists and appending them to the recursive
;;; result in order, it only constructs the result one element at a time.  The
;;; interleaved order is achieved by “switching” the arguments list order in the
;;; recursive call.
;;;;;;;;;

(defun interleave (list1 list2)
  (if (null list1)
       list2
       (cons (car list1) (interleave list2 (cdr list1)))
   ))
```

Problem 3

Write a function called swap which takes a list and swaps every two elements in the list. This function should work for any list regardless of what it contains. It should work as follows:

```
? (swap '(1 2 3 4 5))
(2 1 4 3 5)
? (swap '("hi" 2 "bye" 3))
(2 "hi" 3 "bye")
```

Solution

```
;;;;;;;;;
;;;Swap
;;;;;;;;;
;;;This function takes a list and swaps every two elements in the list
;;;;;;;;;

(defun swap (list)
       (if (or (null list) (null (cdr list))) list
              (cons (second list) (cons (first list) (swap (cddr list))))))
```

The only “tricky” part of the problem is the base case. If the list is empty or has only one element, there aren’t enough elements left to do a real swap, so you just return what is left. It would be a mistake to only code the base case to deal with (null list), because for lists of odd length, you will try to take the “second” of a one-element list.

Problem 4

Write a function called `list-rotate` which accepts two arguments: a list and a count. `list-rotate` should return a new list representing the result of rotating the elements of the original list the specified number of times. The list should be rotated to the left if the count is positive, or to the right if the count is negative. The function should work as follows:

```
> (list-rotate '(1 2 3 4 5) 0)
(1 2 3 4 5)
> (list-rotate '(1 2 3 4 5) 4)
(5 1 2 3 4)
> (list-rotate '(1 2 3 4 5) -2)
(4 5 1 2 3)
> (list-rotate '(1 2 3 4 5) -11)
(5 1 2 3 4)
>
```

Solution

```
;;;;;;;;;;;
;;; Function: list-rotate
;;;;;;;;;;;
;;; list-rotate accepts a list and a count, and returns a
;;; a new list representing the result of rotating the elements
;;; of the original list the specified number of times. Note the
;;; use of reverse to consolidate the three different cases to two.
;;; reverse is a built-in routine, but you might consider how
;;; this could be written from scratch.
;;;;;;;;;;;

(defun list-rotate (list count)
  (cond ((zerop count) list)
        ((> count 0) (list-rotate-helper list count))
        (T (reverse (list-rotate-helper (reverse list) (* -1 count))))))



;;;;;;;;;;;
;;; Function: list-rotate-helper
;;;;;;;;;;;
;;; list-rotate helper is enlisted by list-rotate to do the
;;; real work.
;;;;;;;;;;;

(defun list-rotate-helper (list count)
  (if (zerop count) list
      (list-rotate-helper (append (cdr list) (list (car list))) (1- count))))
```

Another simpler solution passes count modulus the length of the list to `list-rotate-helper`:

```
(defun list-rotate (list count)
  (list-rotate-helper list (mod count (length list))))
```

Problem 5

There is a nifty LISP function named `max` which takes an unlimited number of reals as arguments, and returns the largest one.

```
? (max 4 7)
7
? (max 1 2 3 4 5 6 5 4 3 2 1)
6
```

Let's write a similar version of `max` that, instead of taking any number of arguments, takes a list that (possibly) has nested lists inside and returns the largest real number anywhere in the whole nested structure, like

```
? (max-list '(4 7))
7
? (max-list '(1 2 (3 4) 5 ((6) 5) 4 (3 2 1)))
6
```

You may assume that the empty list `'()` does not appear anywhere in the list. You may assume that all the elements in the list are numbers or lists (i.e. no strings or symbols will be within the list)

Solution

```
(defun max-list (list)
  (cond ((atom list) list)
        ((null (cdr list)) (max-list (car list)))
        (T (max (max-list (car list)) (max-list (cdr list)))) ) )
```

Problem 6

Let's define the depth of a list to be the largest number of unbalanced left parentheses that one can find to the left of an atom. We want to write a function called `max-depth` which returns this number. Feel free to make use of the functions `max` and `1+` in your solution. `1+` is a cool little function which returns the successor of an integer.

```
? (1+ 3)
4
? (1+ (* 4 4))
17
? (max-depth '())
0
? (max-depth '(1 2))
1
? (max-depth '(1 (2) ((3)) (((4))) ((((5)))) (((((6)))))))
6
? (max-depth '(() (1 3 (2 3 (6) 5 (6 (4 0) 2 (7 (3 8 ((3)) 9))) (1) (2 2))) ()))
8
?
```

Solution

```
;;;;;;;;;
;;; Function: max-depth
;;;;;;;;;
;;; The depth of an empty list is defined to be zero, as there are no
;;; atoms other than the empty list itself.  If the first element of the
;;; list is an atom, then we know that the depth of the list is at least
;;; one, but it could be greater if the depth of the rest of the list is greater
;;; than one.  Finally, if the first element of the list is itself a list, then
;;; we need to recursively determine the max-depth of the first element and compare
;;; it to the max-depth of the rest of the list.
;;;
;;; You know a language is powerful when the comments are more spacious than the code.
;;;;;;;;;

(defun max-depth (list)
  (if (null list) 0
      (if (atom (car list)) (max 1 (max-depth (cdr list)))
          (max (1+ (max-depth (car list))) (max-depth (cdr list))))))
```