Programming Challenge: Recursive List Processing and HOFs

This assignment affords you an opportunity to do some simple recursive list processing. It is written in the voice of a lab for a programming course. The trace function is featured throughout the assignment.

Learning Outcomes

Upon successful completion of this challange it is anticipated that you will be able to:

1. Define some functions in Lisp.
2. Write some recursive programs.
3. Make use of some higher order functions that apply to lists.
4. Perform some classic list processing.
5. Use `trace`, Lisps premier monitoring and debugging tool.

Instructions

Perform the tasks, in the specified order, defining all of the functions in a file called `lp.l`, preferably in a folder called `library` situated just within your Lisp programming directory.

Task 1: singleton-p – rac – rdc – snoc – palindrome-p

1. Define the functions `singleton-p`, `rac`, `rdc`, `snoc`, and `palindrome-p`, **the latter four recursively**, as presented in class.
2. Engage in a Lisp session which:
   (a) Loads the `lp.l` file.
   (b) Demos the `singleton-p` function three times, on a list of length 1, on a list of length 2, and on a list of length 7.
   (c) Sets the trace flag for the `rac` function and runs it twice, once on a list of length 1 and once on a list of length 4.
   (d) Sets the trace flag for the `rdc` function and runs it twice, once on a list of length 1 and once on a list of length 5.

(e) Resets the trace flag (untrace) for `rac` and `rdc`.

(f) Sets the trace flag for the `snoc` function and runs it three times, adding `BLUE` to (1) the empty list, (2) the list containing just the atom `RED`, and (3) a list containing four blue-ish colors.

(g) Resets the trace flag for `snoc`.

(h) Sets the trace flag for the `palindrome-p` function and tests it on each of the following lists: `( )`, `( PALINDROME )`, `( CLOS SLOC )`, `( FOOD DRINK FOOD )`, `( 1 2 3 4 5 4 2 3 1 )`, and `( HEY HEY MY MY MY MY HEY HEY )`

(i) Resets the trace flag for `palindrome-p`.

3. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 2: select – pick

1. **Using basic recursive list processing** (i.e., without using `nth`, or any interative construct), write a function called `select` taking one argument, presumed to be a non-negative integer, which returns the element of the list in the given position, assuming that it exists. In other words, *simulate* `nth`.
2. Write a function called `pick` which takes a list as its sole argument and uses `select` to pick a random element from the list.
3. Engage in a session which:

   (a) Loads the `lp.l` file.

   (b) Sets the trace flag for the `select` function and demos the `select` function on at least three representative lists.

   (c) Resets the trace flag for the `select` function.

   (d) Demos the pick function on at least three representative lists.

4. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 3: sum – product

1. Write the function `sum` as presented in class, and the analogous function `product`.
2. Engage in a session which:

   (a) Loads the `lp.l` file.

   (b) Sets the trace flag for both the `sum` function and the `product` function.

(c) Runs each of these functions on the lists: `()`, `( 496 )`, `( 1 11 111 )`, and `( 1 2 3 4 5 6 7 8 9 10 )`.

3. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 4: iota – duplicate

1. Write the function `iota` as presented in class, and then then write the **recursive** function `duplicate` taking parameters `n` and `lo` which generates a list containing `n` instances of `lo`, where `n` is a nonnegative integer and `lo` is a Lisp object.
2. Engage in a session which:
   (a) Loads the `lp.l` file.
   (b) Sets the trace flag for both the `iota` function and the `duplicate` function.
   (c) Uses `iota` to generate `(1)`.
   (d) Uses `iota` to generate `(1 2 3 4 5 6 7 8 9 10)`.
   (e) Uses `duplicate` to generate `(boing boing boing)`.
   (f) Uses `duplicate` to generate `(9 9 9 9 9 9 9 9 9)`.
3. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 5: factorial – power

1. Write the function `factorial` using `iota` and `product`. Write the function `power` using `duplicate` and `product`.
2. Engage in a session which:
   (a) Loads the `lp.l` file.
   (b) Evaluates the following Lisp forms: `( factorial 5 )`, `( factorial 10 )`, `( power 2 16 )`, and `( power 5 6 )`.
3. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 6: filter-in – filter-out

1. **Using recursion**, write the function `filter-in`, which takes a predicate and a list as parameters, and returns the list of elements which are true with respect to the predicate.

2. **Using recursion**, write the function `filter-out`, which takes a predicate and a list as parameters, and returns the list of elements which are false with respect to the predicate.
3. Engage in a session which:
   (a) Loads the `lp.l` file.
   (b) Runs `filter-in` on three rather different functions.
   (c) Runs `filter-out` on three rather different functions.
4. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 7: take-from

1. **Using recursion**, write the function `take-from`, which takes an object and a list as parameters, and returns the list with all occurrences of the object removed from the list. Constraint: Do not use `remove` or `delete`.
2. Engage in a session which:
   (a) Loads the `lp.l` file.
   (b) Runs `take-from` on three rather different sets of parameters.
3. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 8: random-permutation

1. Write the function `random-permutation` which takes a list as its sole parameter and returns a random permutation of the given list. In doing so, translate the following pseudocode to Lisp:

```
to create a random permutation of list x do
  if ( x is empty ) then
    return the empty list
  else
    let element be a randomly selected element of x
    let remainder be the list with element (one occurrence) removed
    return the list consisting of x followed by a random permutation of the remainder
  end
end
```

2. Engage in a session which:
   (a) Loads the `lp.l` file.

(b) Runs `random-permutation` on five different lists.

(c) Runs it one more time on a reasonable list with its trace flag set.

3. Save the code for these functions and the Lisp session in the part of your solution document dedicated to this task.

Task 9: Mapping Examples

Type the four `mapcar` example expressions provided in the "MAPCAR" section of the lesson on mapping functions into the REPL. Do the same for the example `mapcan` expression in the "MAPCAN" section.

Save REPL interaction in the part of your solution document dedicated to this task.

Task 10: Mapping Exercises

Within the REPL, do all of the parts of "Mapping Exercise 1" from the lesson on mapping functions.

Within the REPL, do all of the parts of "Mapping Exercise 2" from the lesson on mapping functions. (Note that you will have to load the `lp.l` file to do this.)

Save REPL interaction in the part of your solution document dedicated to this task.

Task 11: Lisp Exercises

In a file called `ditties`, write the functions specified in "Lisp Exercise 1", "Lisp Exercise 2", and "Lisp Exercise 3".

Generate a demo which liberally illustrates the behavior of each of these three functions.

Save the code and the demo in the part of your solution document dedicated to this task.

Task 12: Post Your Solution Document to Your Web Work Site

In the appropriate spot on your work site, reference your solution document for this programming challenge.

Due Date

Friday, September 30, 2022, **or soon after that.**