COSC 2P90

Assignment 3 – The LISPening!

Due: Monday, November 11th @5:00PM

Being a classic example of a *functional* language, LISP is an important language for this course. This assignment is broken up into two halves. The first half is a worksheet style, designed to guide you through some of the quirks of LISP. The second half (the 'real' assignment) consists of a single task, for which you must create a generalized recursive solution.

Part The First

Complete and answer the following. For questions where I ask you of the difference between two statements, or what happens when you run a piece of code, I am NOT simply asking what appears on the screen. ***Actually explain it***.
Also, please remember that, when I say that a function should *return* a value, it needs to actually ***return*** a value. Simply printing is not sufficient.

1. What is the difference between the lines:

```
(print (+ 1 1))
```

and

```
(print '(+ 1 1))
```

2. What is the difference between DEFVAR and DEFPARAMETER?

3. Show me three different ways to create the list: (1 2 3)

4. Write a "random" number-generating function, `rand`, that accepts no parameters and ALWAYS returns the value 4.

5. Write a function, `dozenify`, that, when given a number (as its sole parameter), will return `"dozen"` if it's 12, or `"not dozen"` if it isn't.

6. Write a function, `frenchify`, that, when given a number, returns `"un"` if it's 1, `"deux"` if it's 2, `"trois"` if it's 3, and `"je ne sais pas"` otherwise.

7. Write a function, `modulizer`, that accepts two parameters (it'll be two lists), and returns a list containing the remainders from dividing each number in the first list by the value in the corresponding position in the second list.

e.g. `(modulizer '(4 5 6) '(1 2 3))`
result: `(0 1 0)`

8. State the output(return) of the following code:

```
(let ((x '(1 2 3))) (format t "~a~%" x) (push 4 x) (format t "~a~%" x))
```

9. Consider the two following functions:

```
(defun wibblywobbly(x) (push 5 x) (format t "~a~%" x))
(defun timeywimey(x) (let ((y x)) (push 5 y) (format t "~a~%" x)))
```

Assuming each is intended to act on a list, what is the difference between them? (Make sure to include the WHY!)

10. Suppose you have a global variable, `*x*`, that has the list `((1 2 3) (4 5 6) (7 8 9))`.
What is the result of `(nth 1 *x*)`?

Part The Second

Premise:

A: "Hey Claire! I was supposed to start my break 17 minutes ago!"
C: "Yeah, I know, Abby, but look at that line you've got at your register! You can't just abandon them all!"
A: "I fail to see why not. In any event, you're the manager, so isn't it your job to handle things like that?"
C: "Fine! I'll open a till on the third register. But we're still going to have to get all those customers over here."
A: "And that's difficult... why? Just tell everyone to move over to your checkout."
C: "Yeah, see, that's the problem. All of those customers are computer scientists, here for our Tech and Algorithms sale."
A: "... seriously?"
C: "Absolutely. They love the stuff. Unfortunately, as computer scientists, they have certain... quirks..."
A: "I'm going to wish I'd simply started my break early, aren't I?"
C: "Oh yeah. See, computer scientists are very picky people. They don't exactly go with the flow. If you told everyone to just swamp over to another checkout, they wouldn't end up in the same ordering as they started, so some customers would effectively have others cutting in front of them."
A: "I think I can see where this is going... and, having their heads in the clouds, they don't really ever seem to notice you talking unless you address them personally, so you can really only direct one person at a time, right?"
C: "Precisely! And, if you start telling them to just 'go to the back of this line', they'll just wander off, so you have to keep them where you can see them."
A: "So then, what? You just keep calling the first person from my line, and adding them to the front of your line?"
C: "No! That would reverse the line! We'll have to get Betsy to help us here."
B: "Oh, don't drag ME into this! I'm already on MY break! Having me a sammich!"
C: "Relax, Betsy. I'll be the only one actually doing work. Computer scientists are relatively docile. We can keep shifting them back and forth indefinitely and they'll just accept it."
A: "Okay, so I finally get it! We just move everyone from my line into Betsy's, reversing the line, and then from hers to yours, restoring order! That doesn't sound so bad!"
C: "Ha! Like I said, PICKY! Not to mention obstinate! No computer scientist could stand being forced in line behind someone they were previously ahead of for even one second! No... this is going to call for some desperate measures."
B: "Ah heck, are you talking about what I think you're talking about?"
A: "I think she is. If only we had a language like LISP, to make that recursive algorithm easier..."
C: "Indeed. We're certainly going to pretend that LISP somehow makes this problem easier to solve!"

Your Task:

The (slightly wordy) dialog above describes a basic queueing problem. You'll be writing a LISP program that, when given a list of names, will shift all of those names, one-by-one, from one position to another, with an additional one being available to help with the transition. You can think of them as starting at A, ending up at C, and having B for additional storage (though it may be easier to code them as 0, 2, and 1, respectively).
However, you need to adhere to the following rules:

- Whenever you move a name, you need to announce which name you're moving, which line you're moving from, and which line you're moving to.
- You may only move one name at a time. You may, however, move the same name an arbitrary number of times.
- Any name you move for a step must be at the FRONT/TOP of its line.
- Whenever a name is moved, in the new line it must stay **in front** of all names it was in front of in the original line, and **behind** all names it was behind in the original line. Clearly, this will severely limit the legal moves available to you.

Any solution violating a single one of those four rules will instantly be awarded a zero, irrespective of what else is on the marking scheme.

Hints:

- If you look at this carefully, it's actually a pretty old problem. ...actually, you probably don't even need to look all that carefully to spot it. That said, don't let that tempt you into plagiarizing. It'll be caught. Fer real.

- Consider defining more than one function. Personally, I used three (one to do the actual work, given three lists, a user-friendly one to accept a single parameter of the original list and invoke the 'real' one, and one to convert list indices into “Abigail”, “Betsy”, and “Claire”).
- I very strongly suggest that you develop this incrementally. First make sure that you know how to shift a single value from one list to another. Only then should you start considering things like recursion.
- The sample execution I provide below finishes off by showing the final state of the three lines. This isn't at all necessary; but is handy for easily seeing what happened.

Sample Execution:
```
[1]> (hannah '(Phil Ned))
Move PHIL from ABIGAIL to BETSY
Move NED from ABIGAIL to CLAIRE
Move PHIL from BETSY to CLAIRE
(NIL NIL (PHIL NED))
```

Using LISP on Sandcastle

You are more than welcome to install some flavour of LISP on your own computer, but it's also available on Sandcastle. Since the marker will probably be grading on Sandcastle, if you do install it at home, you may wish to use the same variety that he'll be using (CLISP). It isn't likely to make a difference, but it's up to you to verify that he can execute your submission.

To execute a LISP script, simply include it as a command-line parameter. However, in order to make use of the REPL (Read-Eval-Print Loop. i.e. the prompt), you'll need to add `-repl` first.

e.g. `clisp -repl myscript.lsp`

This will execute the contents of the script file, `myscript.lsp`, (alerting you of any errors it catches), including definitions of functions, and then give you the normal LISP environment.

Important Tip

As mentioned earlier, the first part is written to help guide you through learning some basic tricks in LISP. However, you'll still need a proper reference. There are *tons* of great references online (just google 'Common Lisp'), but I'm going to suggest this one:
`http://www.gigamonkeys.com/book/`
It's got some pretty good stuff in it.

Submission

Physical:

Print your solution for the first part. Print your source code for the second part, as well as a sample execution with at least four names. Staple all of these together, attach a departmental coverpage, and drop it into the course dropbox in J-Block by the date and time specified above.

Electronic:

Put your code into a separate folder on sandcastle (your Z: drive). SSH/PuTTY in, navigate to that folder, and type:
`submit2p90`
It should prompt you to specify things like assignment #, etc.