Assignment 1: Lisp Practice

UMaine COS 470/570 – Fall 2021

Assigned: 8/30/2021 *Due: 9/10/2021*

Lisp is one of the primary languages used in AI, and much of the homework in this class will be in Lisp. Aside from this, there are other reasons to learn (or review) Lisp. For one thing, many programs and other languages use concepts and, more often, syntax from Lisp, and some programs use Lisp itself. Also, learning different languages, especially ones that are quite different from ones they know, is a very good thing for computer science majors—and it's required for accreditation. And, finally, if past experience is any guide, some of you might find that you fall in love with the language and want to keep using it.

But don't take my word for it; here's what others have said about Lisp (from wikiquote[1] via CLiki[2].):

[1] `http://en.wikiquote.org/wiki/Lisp\_programming\_language`

[2] `https://www.cliki.net`

Getting started

The first hurdle to using Lisp is finding and installing a Lisp interpreter. See the "Lisp Packet" on BrightSpace for detailed directions on installing and running Lisp.

With respect to programming in Lisp, I will schedule and/or record a two-hour Lisp basics session later this week or next week to help you become familiar with the language. There are many on-line resources for learning Lisp; some of these are listed in the Lisp Packet document.

The Assignment

For this assignment, you will do each of the following exercises. Your functions must be contained in a file named `"assignment1.lsp"`. You will load that file (see Lisp's `load` function), then call each of the functions to test them and to show that they work. Put the results of doing this in another file, `"results.txt"`. Turn in both files together on BrightSpace as a zip file or a gzip'd tarfile. (Note: to capture the results, see the Lisp function "dribble", described at this link[3]; note that the webpage linked, from Steele's *Common Lisp: The Language* (2nd ed, known as CLtL2), also describes some useful debugging tools.)

1. `first2`: Simple list manipulation.

   Write a function, `first2`, that takes a list as its argument and returns the first two elements of that list as a new list. Note: this should return a list, *not* a dotted pair. For example,

   ```
   > (first2 '(a b c d e f))
   (A B)
   ```

   In this example and those to follow, `>` represents the Lisp interpreter prompt, with the line after being the result of evaluating the Lisp expression after the prompt. The prompt from your Lisp will probably be different.

2. `add1`: Simple math.

   Write a function, `add1`, that takes an integer as an argument and returns that integer plus 1.

3. `listadd`: Recursion and iteration.

   (a) Write a function, `listadd1`, that will add 1 to each element in a list of integers:

   ```
   > (listadd1 '(2 3 4 5))
   (3 4 5 6)
   ```

   This function must use recursion to do this.

   (b) Write another function, `listadd2`, that uses `mapcar` to do the same thing.

   (c) Write another function, `listadd3`, that uses one of Lisp's loop constructs (e.g., `dolist`, `dotimes`, or `loop`) to do the same thing.

4. `flatten`: Complex recursion.

   Write a recursive function, `flatten`, that takes as an argument a list that can include embedded lists and returns a list in which the embedded parentheses have been "removed"—i.e., a list whose elements are all of the elements of all of the embedded lists. For example,

   ```
   > (flatten '(a (b c) d (e (f (g (h) i))) j))
   (A B C D E F G H I J)
   ```

[3] `https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node230.html`

5. `last2`: More complex list manipulation.

   Write a function, `last2`, that takes a list as its argument and returns the *last* two elements, e.g.:

   ```
   > (last2 '(a b c d e))
   (D E)
   ```

6. `my-reverse`: More complex list manipulation and recursion.

   Write a function, `reverse`, that reverses the order of the elements in a list:

   ```
   > (my-reverse '(a b c 1 2 3))
   (3 2 1 C B A)
   ```

   You must use recursion, and you may *not* use the built-in `reverse` function!

7. `printlist`: Using formatted output.

   Lisp has a function, `format`, for producing formatted output that is similar to C's `printf` facility and somewhat akin to Python's formatted print functionality. Use the `format` function to write a function, `printlist`, that takes a list as an argument and produces the following output:

   ```
   > (printlist '(1 2 3 4 ... 100)) ;; "..." here replaces the actual elements
   First element: 1
   Second element: 2
   Third element: 3
   Fourth element: 4
   Fifth element: 5
   Sixth element: 6
   Seventh element: 7
   Eighth element: 8
   Ninth element: 9
   Tenth element: 10
   Eleventh element: 11
      ...
   Ninety-ninth element: 99
   One hundredth element: 100
   NIL
   ```

   This should work for any list, with any number of elements (which need not be numeric). Note: the words "first"', "second", etc., are ordinal numbers generated by `format`. Also note that `format` should be used to automatically capitalize the ordinal numbers.

   In order to do this problem, you'll need documentation for the `format` statement. The definitive source is Common Lisp: The Language[4] (2nd ed.) by Guy Steele.

8. `copyfile`: File IO.

   Write a function, `copyfile`, that takes two file names as arguments and copies the contents of the first file to the second. You cannot simply use Lisp's shell interface to get the operating system to do it for you.

[4] `http://www-2.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`

As in any programming language, you'll need to open and close the files. In Lisp, the easy way to do this is with the `with-open-file` special form; for example, to open a file "foo" and read all its contents, you could do:

```
(with-open-file (in "foo" :direction :input)
  (loop with line
      with i = 1
      until (eql :eof (setq line (read-line in nil :eof)))
      do (format t "Read line ~s~%" i)
 (incf i)))
```

In this, `in` is a stream variable (which doesn't have to have the name "in") that is read from. This form is also used to write to files, with some slight changes, e.g.:

```
(with-open-file (out "bar" :direction :output :if-exists :supersede
 :if-does-not-exist :create)
  ...)
```

You can use `read` to read an entire Lisp expression and `write` or `print` to write such an expression. You can read a line at a time with `read-line` and write an entire line with `write-line`.

9. Create a CLOS (Common Lisp Object System) class `animal`. Create two more, `bird` and `cat`, and make them subclasses of `animal`. The `animal` class should have an instance variable, `num-legs`, that is initialized to 4. `bird` should override this with the value 2. There should be a method `speak` of `animal`. Using only method definitions, when `speak` is called on an instance of `bird`, the value "tweet" should be printed, while if called on `cat`, the value "meow" should be printed. After defining these, do the following:

```
1  (setq c (make-instance 'cat))
2  (setq b (make-instance 'bird))
3  (describe c)
4  (describe b)
5  (speak c)
6  (speak b)
```

10. Using *no global variables*, define a function `next` that has the following behavior: If called with an argument that is a list, it returns nothing, but the *next* time it is called, without an argument, it returns the first element of the list. If called without an argument, it returns the next item on whichever list was passed to it most recently. The behavior looks like this:

```
> (next '(a b c d))
NIL
> (next)
A
> (next)
B
> (next)
C
```

```
> (next)
D
> (next)
NIL
> (next)
NIL
> (next '(1 2 3))
NIL
> (next)
1
```

Show the output on different lists than those used in the example.
Hint: How would you do this in Python? Maybe with a *closure*?