Lisp Warmup

Thursday, February 18, 2021 3:25 PM

Self-graded Homework 1 for Emacs and Lisp
DUE Feb 18◊ th 11:55PM Estimated time 3-4 Hours.

Can you login or ssh to a CS department machine or use ter? Or can you install gnu emacs and sbcl on your laptop? The office hours calendar is linked from Latte if you need help.

Can you bring up **emacs** Editor? type **emacs or emacs –nw** if you don't run xwindows.

If you have never run emacs, do the tutorial **^ht**. At least you should be able to move around with the cursor keys or **^f ^b ^p and ^n**

Can you do **Esc x Run-lisp** to open an inferior shell lisp window?

If it fails you need to edit your .emacs file. Do **^x^f~/.emacs** to open the file and add the line **(setq inferior-lisp-program "/usr/local/bin/sbcl")** or the actual path**.**
Then do **^x^**s to save and **^x^c** to exit. Then run emacs again and Esc-x run-lisp should work.

At this point, you can play in lisp. Type () and hit return and the REPL should return nil. use **Alt-p** and **Alt-n** for the history mechanism.

To see all commands for INFERIOR LISP MODE, run **^hm, or print out the cheatsheet.**

Can you do **^x2** to split the screen?

Can you do **^x^f** and open a file called test.lsp. The filetype is important as emacs uses it to set the buffer into lisp mode. **^hm** again to see the commands.

In that buffer (You move back and forth with **^xo**) you can define your lisp functions and global variables and play.

1) Try writing factorial: **(defun fact (n) (if (< n 2) 1 (* n (fact (- n 1)))))**

2) now send it from your file buffer to the inferior lisp evaluator by typing **^c^e**

3) can you also define fibonacci and send it over with ^c^e:
(defun fib (n) (if (< n 2) 1 (+ (fib (- n 1) fib (- n 2)))))

4) now when **^xo** or click into the lisp buffer and run **(fact 10)**. it works. **(Fact 100)** shows you that lisp works with "bignums", useful for number theory and cryptology.

5) When you run **(fib 2)** you get into trouble and land in the debugger, which is supposed to be self-documenting. Usually typing **ABORT** or a number gets you out.

6) Fix up **fib** so it works. Hint: Check the parentheses.
7) Running **(FIB 10)** and **(FIB 30)** a few times make you confident to do something ridiculous like **(fib 100)**. Luckily **^c^c** interrupts the lisp process, and you can abort it.

Congratulations, you are now using LISP within Emacs. Now to have some lisp practice!

8) CONS and LIST practice. What will be the result of these expressions? (Guesses typed, wrongs highlighted)

a) `(CONS A B) => (<A> . <B>)`
b) `(CONS 'A 'B) => (A . B)`
c) `(CONS 1 2 3) => Error! Cons must have exactly 2 arguments!`
d) `(LIST 'A 'B) => (A B)`
e) `(CAR '(A B C)) => A`
f) `(CDR '(A B C)) => (B C)`
g) `(CADR '(A B C)) => Error! Can't find CDR of A`
   a. `CADR is evaluated right to left (like expressions, so it's actually (CAR (CDR '(A B C))), which is B`
h) `(CONS 'A (LIST 'B 'C 'D)) => (A B C D)`
i) `(CONS '(A B C) 'D) => ((A B C) . D) OR ((A .(B . (C . NIL))) . D)`
j) `(CONS '(ONE TWO THREE) '(FOUR)) => ((ONE TWO THREE) . (FOUR))`
   a. `((ONE TWO THREE) FOUR)`
k) `(APPEND '(1 2 3) 4) => (1 2 3 4)`
   a. `(1 2 3 . 4) [Pretty sure that is a list with three elements, the last of which is 3 . 4, and I have no idea why on Earth it does this.]`
l) `(APPEND (LIST 1 2 3) (CONS 4 NIL)) => (1 2 3 4)`
m) `(SETF FOO1 '(1 2 3 4 5 6 7 8 9 10))`
   a. `This complains about an undefined variable, but works`
n) `(NTH 0 FOO) => 1`
o) `(NTH 10 FOO) => ERROR!`
   a. `Technically NIL, but whatever`
p) `(SETF FOO2 (LOOP FOR I FROM 1 TO 10 COLLECT I)) => (1 2 3 4 5 6 7 8 9 10)`
q) `(EQ FOO1 FOO2)=> T`
   `a. NIL`
r) `(EQUAL FOO1 FOO2) => T`

Formula Translation (That's how FORTRAN got its name!)

1) Convert between infix and prefix (LISP).
   a. 2 + 2 * 5 + 5 => (* (+ 2 2) (+ 5 5))
   b. (B2 + 4*A*C) / 2*A => (/ (+ B2 (* 4 (* A C))) (* 2 A))
   c. (SQRT (+ (* 5 6)(* 3 2))) => sqrt(5 * 6 + 3 * 2)
   d. (= E (* (* M C)(* M C))) => E = (M * C) * (M * C)

2) **EQ EQL EQUAL** and = are different. How so and why?
   EQUAL compares values, EQ compares memory addresses, EQL does EQ but isn't stupid when it comes to strings and numbers.
   * I had to check the docs/Stack Overflow for EQ vs. EQL.

3) **(rrotate '(a b c d e))** should return (e a b c d) while **(lrotate '(a b c d e)** -> (b c d e a)
Write each function and explain why you used various functions such as **nth, butlast, cons,** or **append**.

```
(defun rrotate (lst)
  (APPEND (last lst) (butlast lst)))

(defun lrotate (lst)
  (APPEND (cdr lst) (LIST (nth 0 lst))))
```

Both functions split the list into two sublists: the stuff to add to the front, and the stuff that goes in the back (with the stuff that's moving to the front already removed). Rrotate uses last and butlast for this, while lrotate uses cdr (effectively butfirst) and nth 0 (effectively first). Then, append is used to merge the lists in the right order.

Recursion Practice

To write a recursive function you need a base case and a test for that case, a way to reduce the current value closer to the base case, and the function to increment or collect the partial computation.

To add up the numbers from 1 to N, the base case is 1, the reduction is -1, and the increment is the current value:

```
(defun sum2n (n)
    (if (= n 1) 1
         (+ n (sum2n (- n 1)))
Of course, simply calculating n(n+1)/2 would be constant time😊
```

1) Write LENGTH, REVERSE, and APPEND recursively.

```
(defun arilength (lst)
  (if (null lst) 0 (+ 1 (length (cdr lst)))))

(defun arirev (lst)
  (if (< (length lst) 2) lst
    (append (last lst) (butlast (cdr lst)) (list (nth 0 lst)))))

(defun ariappend (lst itm)
  (if (atom itm) (ariappend lst (list itm))
     (if (null lst) itm (ariappend (butlast lst) (cons (car (last lst)) itm)))))
```

2) (defun EXPO n k) returning N to the Kth power. To make it faster, consider that instead of k steps of multiplication, x^{2y} is $x^y x^y$

```
(defun a_expo (n k)
  (if (= K 0) 1 (* n (a_expo n (- k 1)))))

(a_expo 5 3000)

(defvar val 1)
```

```
(defun a_expo_fast (n k)
 (cond ((= k 0) 1)
      ((= (rem k 2) 0) (setq val (a_expo_fast n (/ k 2))) (* val val))
      (T (* n (a_expo_fast n (- k 1))))))
```

3) SquareRoot is a little harder. You recurse with the number x and a guess. If |x-guess^2| < .0001 return the current guess, otherwise recurse with x and the average of guess and x/guess. Use the &optional (guess 1.0) for the starting call.

```
(defun a_sqrt (n &optional (guess 1))
 (if (< (abs (- n (* guess guess))) 0.0001) guess (a_sqrt n (/ (+ guess (/ n guess)) 2))))
```

Here are my Factoring functions from lecture:

```
(defun factor (n &optional (i 1))
       (cond ((> i (sqrt n)) nil)
         ((zerop (rem n i))(cons (/ n i)
                                (cons i (factor n (+ 1 i)))))
         (t (factor n (+ i 1)))))

(defun factor1 (n)
  (loop for i from 1 to (floor (sqrt n)) append
    (if (zerop (rem n i)) (list i (/ n i)) nil)))
```

Factoring 36 and other squares with my simple code results in a duplicate return value. There are 2 obvious ways to repair this:

4) Test if I is the sqrt of N and only collect it once. Modify the code to fix it this way. Remember that **COND** works like a series of ifelse's.

```
(defun factor (n &optional (i 1))
 (cond ((> i (sqrt n)) nil)
       ((= i (sqrt n)) (list (sqrt n)))
          ((zerop (rem n i))(cons (/ n i)
                    (cons i (factor n (+ 1 i)))))
          (t (factor n (+ i 1)))))
```

5) Write a **REMDUP** program to remove duplicates from a list. Write **remdup** two ways, recursively, and using the LOOP macro. After testing, redefine **FACTOR** with remdup.
I didn't do this one.

Lambda is a magic form which defines a "no-name" or instant function which can be used like any other function, and can be passed into things like sort, member, and function mappings. The syntax is **(LAMBDA (PARAMETERS) (EXPR)(EXPR)*),** and to use a lambda function you quote it with the function quote #'. For example, to square each member of a list you can do **(MAPCAR #'(LAMBDA (X)(* X X)) '(1 2 3)) -> (1 4 9).**

6) Other languages have FOR and WHILE loops. Lisp has "Mapping" functions. Use mapping functions to
   a. Reverse each sublist ((a b c)(d e f)) → ((c b a)(f e d))
   b. Extract the second element of each sublist ((1 2 3)(4 5 6)) -> (2 5)

c. Sum each sublist ((1 2 3)(4 5 6)(7 8 9)) -> (6 15 24); hint, you need a lambda and apply
d. Remove odd numbers from a list (1 2 3 4 5 6)->(2 4 6)
    ;hint mapcan appends results so either return (2) or NIL from the lambda function

```
(mapcar #'(LAMBDA (lst) (arirev lst)) '((1 2 3) (4 5 6)))
(mapcar #'(LAMBDA (lst) (car (cdr lst))) '((1 2 3) (4 5 6)))
(mapcar #'(LAMBDA (lst) (apply #'+ lst)) '((1 2 3) (4 5 6) (7 8 9)))
(mapcan #'(LAMBDA (n) (and (= (rem n 2) 0) (list n))) '(1 2 3 4 5))
```

7) You've heard of Map-Reduce, the paradigm for many Google products? Use **reduce** to
   a. Do factorial **(reduce #'??? '(1 2 3 4 5)** -> 120
   b. Flatten a top-level list **(reduce '#??? '((1 2 3)(4 5 6)(7 8 9)))** -> (1 2 3 4 5 6 7 8 9)

```
(reduce #'* '(1 2 3 4 5))
(reduce #'append '((1 2 3) (4 5 6) (7 8 9)))
```

8) Play with the Loop Macro:
   a. Loop using sum to sum the elements of a list
   b. Loop using ON to calculate successive products (f '(1 2 3 4))-> (1 2 6 24)
   c. Loop using FROM and TO to collect a list from 1 to N
   d. Loop using FROM, TO, and BY to add the even numbers 2 thru 20
   e. Loop using FROM and DOWNTO to get a list from N downto 1
   f. Loop using COUNT how many even numbers in a list
   g. Make sure ALWAYS the numbers in a list are odd.
   h. Using factor to test for a prime number, Loop using THEREIS to find the first prime after input N
   i. Loop with two variables using AS (loop for I from 1 as l in '(a b c)…
   j. Loop with a local variable using WITH

```
(loop for x in '(1 2 3 4 5) x sum x)
(reverse (LOOP for x on '(4 3 2 1) collect (reduce #'* x)))
(loop for x from 1 to 10 collect x)
(loop for x from 2 to 20 by 2 sum x)
(loop for x from 10 downto 1 collect x)
(loop for x in '(1 2 3 4 5) when (evenp x) count x
(loop for x in '(1 3 5) always (oddp x))
(loop for x from 1 thereis (= 2 (length (factor x))))
(loop for x in '(a b c d e) as i from 1 collect (list i x))
(loop for x in '(a b c d e) with num = 42 collect (list num x))
```

That's all. If you want to keep learning lisp, please try some of the problems in Norvig Chapters 1 and 3. Also, http://lisp.plasticki.com has a set of elementary problems by a Dad teaching lisp to his teenagers. If they can do it, so can you!

Most answers will be posted the day after deadline, and you will submit your best estimate of what percentage (out of 100) you accomplished, which will count for 2% of your final grade.