Homework 2: LISP practice

Programming Languages (Fall 2024)

October 24, 2024

This homework is designed to get you used to programming in Lisp and thinking about functional programming. As always collaboration is encouraged but please cite your sources of information (including interaction with other students) as comments in your deliverable.

Deliverables: Submit one file "∼/CS59/hw2.lisp" on the pond which contains the necessary functions defined with the `defun` command. For instance, if this homework was just question 1.1, your file might look something like this:

```
;; Required libraries
(require 'cl-lib)

;; Function sum-odds for Question 1.1
(defun sum-odds (l)
  (if (null l)                ; If we've reached the end of the list, return 0
    0
    (+ (sum-odds (cdr l))     ; otherwise loop down the rest of the list
      (if (cl-oddp (car l))   ; check if the first element of the list is odd
        (car l)               ; if it is, add it to our sum
        0                     ; otherwise don't change the sum
      )
    )
  )
)
```

Note in order to use the `cl-oddp` function the `cl-lib` library needed to be imported. You can include all imports at the top of your file and then the function definitions below them. Writing helper functions for your main functions are encouraged for several of these questions.

Important: All code should work within the emacs-lisp environment running on *thepond*. (ssh into *thepond* and run "emacs", open your file and test it!)

Side Effects: Coming from an imperative world of programming it can be tempting to use side effects in your Lisp programming. These are functions like `setq` which have lingering effects outside of their scope. For this assignment, the only functions with side effects you are allowed to use is `defun` and `defvar` (and you may only use these in the outermost scope).[1]

[1] You may also import libraries and enable `lexical-binding` which require using functions that technically have side effects but this seems like a reasonable break from purity.

Question 1 [35 pts]: Basic LISP practice

Just some simple functions to get you used to working in emacs, testing your code and otherwise working with Lisp. All functions can assume the argument is correctly formatted / what we are expecting. Do not use `cl-reduce` and `mapcar` for parts 1 - 4 of this question.

1. Define a function `sum-odds` which takes one argument, a list, and returns the sum of all the odd numbers inside the list. For example:
   ```
   (sum-odds '())              ; returns 0
   (sum-odds '(4 2 7 3 -8 -1)) ; returns 9
   ```
   One possible implementation of this function is defined above.

2. Define a function `rev-list` which takes one argument, a list, and return the list with it's elements in reverse. For example:
   ```
   (rev-list '())              ; returns nil
   (rev-list '(7))             ; returns (7)
   (rev-list '(1 2 (3 4) 5 6)) ; returns (6 5 (3 4) 2 1)
   ```

3. Define a function `split-parity` which takes one argument, a list of integers, and returns a new list containing two lists. The first sub-list is all the odd elements of the integer list and then second sub-list is all the even elements of the integer list. For example:
   ```
   (split-parity '())                    ; returns (nil nil)
   (split-parity '(7 1))                 ; returns ((7 1) nil)
   (split-parity '(0 1 2 3 4 5 6 7 8 9)) ; returns ((1 3 5 7 9) (0 2 4 6 8))
   ```
   Do not re-order the elements of the list when you split them. For full points, do this with only one pass through the list.[2]

4. Define a function `sum-neighbors` which takes one argument, a list of integers, and returns the sum of all elements in the list which are equal to the element directly before or after it. For example:
   ```
   (sum-neighbors '())                   ; returns 0
   (sum-neighbors '(2 1 2))              ; returns 0
   (sum-neighbors '(3 5 4 4 4 5)))       ; returns 12
   (sum-neighbors '(2 2 3 3 2 2 2 2 2)) ; returns 20
   ```

5. Define a function `rotate-left` which takes two arguments, a list and a number, and returns the list after being *cycled left* the given number of times. For example:
   ```
   (rotate-left '() 2)                ; returns nil
   (rotate-left '(1 2 3 4) 1)         ; returns (2 3 4 1)
   (rotate-left '(1 2 3 4 5 6 7) 14) ; returns (1 2 3 4 5 6 7)
   (rotate-left '(1 2 3 4 5) -2)      ; returns (4 5 1 2 3)
   ```

6. Define a function `flatten-all` which takes one argument, an arbitrary nesting of lists and integers, and returns a list containing all the non-list non-nil elements inside its parameter. For example:
   ```
   (flatten-all '())                           ; returns nil
   (flatten-all 7)                             ; returns nil
   (flatten-all '(2 () 3)                      ; returns (2 3)
   (flatten-all '((1 2) 3 (4 ((5) 6) () 7) 8)) ; returns (1 2 3 4 5 6 7 8)
   ```

7. Define a function `fast-sort` which takes one argument, a list, and returns a sorted version of the list. You should use an efficient (i.e. runtime of $O(n \log(n))$ for a list with n elements) sorting algorithm.[3]
   ```
   (fast-sort '())                    ; returns nil
   (fast-sort '(3))                   ; returns (3)
   (fast-sort '(7 3 0 8 4 6 2 9 1 5)) ; returns (0 1 2 3 4 5 6 7 8 9)
   ```

[2] By this we mean: don't filter the list twice and concat the filtered list together; also don't separate the elements in reverse order and then reverse the two lists

[3] I think merge sort is the easiest, but you can surprise me

Question 2 [20 pts]: Understanding Fold and Map

It is very likely a lot of your previous answers for Question 1 used a lot of recursion mostly to just iterate through the lists. Functional programming doesn't have to involve so much mental effort just to iterate through a list. We have tools such as *fold* (`cl-reduce` in Lisp) and map (`mapcar` in Lisp) which can make some problems a lot easier.

> **Fold:** Fold takes a function, a list and an initial value, and then successively applies this function to the elements in the list starting with the initial value and passing the result of each function onward.
> ```
> (cl-reduce #'/ '(2 4 3) :initial-value 48)   ; returns 2 -- ((48 / 2) / 4) / 3 = 2
> (cl-reduce #'cl-rem '(5 2) :initial-value 7) ; returns 0 -- (7 % 5) % 2 = 0
> (cl-reduce #'cl-rem '(2 5) :initial-value 7) ; returns 1 -- (7 % 2) % 5 = 1
> (cl-reduce #'+ '(3 4 2 1 5))    ; returns 15 -- note the initial-value is optional
> (cl-reduce
>   (lambda (a x) (+ (car x) a)) ; we can define our own functions
>   '((3 2) (4 1) (2 7 8))       ; and iterate over more complicated lists
>   :initial-value 0
> ) ; returns 9 -- ((0 + (car '(3 2))) + (car '(4 1))) + (car '(2 7 8)) = 9
> ```
> There are other optional parameters you can pass to Lisps version of fold: for example you can specify if you want to fold from the end of the list. I encourage you to check out the documents!
>
> **Map:** Map takes a function and a list, and then returns a new list with the function applied to each element of the list.
> ```
> (mapcar #'1- '(2 3 4))          ; returns (1 2 3)
> (mapcar #'null '(2 (3) () () 4) ; returns (nil nil t t nil)
> ```

WARNING: While you might be tempted to fold `or` over a list, this isn't actually possible as `or` is a special form and not an actual function (This is also true of `and`, `if`, `defun`, and several other things that appear in Lisp like they should be functions).[4]

Parts 1 - 4

Do parts 1 through 4 of Question 1 using *fold* and *map* as your only recursive structures. Name the new functions by adding `-norec` to the end of them (e.g. For question 2 part 1, you would write a function called `sum-odds-norec` which has the same behavior as `sum-odds`).

Question 3 [10 pts]: Fast Fibonacci

Consider the following LISP implementation of the Fibonacci sequence:

```
(defun fib (n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))
  )
)
```

This is mathematically elegant and clearly defines the sequence, however this can also be quite slow. You can run this code on *thepond* and it will start to get unbearable to execute around `(fib 32)`. Write a function `fast-fib` which is more efficient. In particular, you should be able to calculate `(fast-fib 750)` in less than a second. Note that EMACS-LISP does not implement tail-recursive optimization because of lexical-binding issues. You should be mindful of your stack space when writing your code. The default recursion depth is 1600, you can do this assignment (and run `(fast-fib 750)`) without needing to change this default. Start by aiming for `(fast-fib 100)` and then seeing how you can optimize from there. How high can you get?[5]

[4] You can always define your own *real function* version of or with a lambda and fold that over a list though!

[5] The best I could do with 30 minutes of thinking was (`fast-fib 94401`), though the error with (`fast-fib 94402`) was an "overflow" error not an "excessive list nesting" error which leads me to believe the numbers are too large for the default settings to handle and not a problem with stack frames... Bonus points if you can rival this and extra bonus points if you can do better!)

Question 4 [15 pts]: Zip

In Python (and many other languages) there is a function called `zip`. Zip takes in an arbitrary number of lists and *zips* them together into one list. The first element of the zipped list is a list containing the first element of all the given lists, the second element of the zipped list is a list containing the second element of all the arguments. If the arguments to zip are of different lengths, then the lists will only be zipped to the shortest length. In LISP this would look something like this:

```
(zip '(1 2 3) '(4 5 6) '(7 8 9)) ; returns ((1 4 7) (2 5 8) (3 6 9))
(zip '(1 2 3 4) '(5 6 7))        ; returns ((1 5) (2 6) (3 7))
(zip '(0) '(1 2) '(3) '(4))      ; returns ((0 1 3 4))
(zip '(1 2) '())                 ; returns nil
(zip)                            ; returns nil
```

Similarly, a list can be *unzipped* too! `un-zip` is a function which takes a single parameter (a list of lists) and returns a list containing the arguments which would've zipped to it's argument (assuming all the lists had the same length... If they didn't then it will return their prefix as there is no way to recover the lost information.

```
(un-zip '((1 4 7) (2 5 8) (3 6 9))) ; returns '((1 2 3) (4 5 6) (7 8 9))
(un-zip '((1 5 9) (2 6) (3 7)))     ; returns '((1 2 3) (5 6 7))
(un-zip nil)                        ; returns nil
```

You will notice, unlike `un-zip` and the other functions you have written for the previous questions `zip` is a little weird: it can take any number of arguments (including zero). You will notice other lisp functions which you have used have this property, for instance:

```
(+ 1 2 3 4 5) ; returns 15
(+)           ; returns 0
```

The way you can implement features like this is with the `&rest` clause of argument lists. You will notice in the complete documentation[6] for emacs-lisp the syntax for argument lists is defined as:

```
(required-vars...  [&optional [optional-vars...]]  [&rest rest-var])
```

So anything you put after a `&rest` clause in an argument list (for a `defun` or a `lambda`) are bound to a single list. As an example, say you only had access to a two-parameter-only add function. Here is one possible implementation of an `add-all` function (Copy it into an emacs buffer and play around with it!):

```
(defun two-param-add (a b)
  (+ a b)                        ; seems pretty straightforward as an add function!
)
(defun add-all (&rest r)
  (if (null r)                   ; if there are no more arguments to add,
    0                            ; we default to zero
    (two-param-add
      (car r)                    ; otherwise we add the first parameter
      (apply #'add-all (cdr r)) ; to the sum of the rest of the elements
    )
  )
)
```

With this knowledge, write both `zip` and `un-zip`.

[6]https://www.gnu.org/software/emacs/manual/html_node/elisp/Argument-List.html

Question 5 [20 pts]: Sieve of Eratosthenes

The *Sieve of Eratosthenes* is an old technique for finding all the prime numbers (and it is still one of the fastest way to find all the prime numbers below some number if that is your goal) . It works by starting with all the numbers from 2 to infinity in a list. You circle the first number (2) and then cross out every number divisible by it (which is easy to do because you just keep adding 2 to the previously crossed out number). You then repeat this process! So 3 will be circled and then you can cross out every 3rd number from there. 5 will be circled and then you cross out every 5th number and so on. The numbers you are left with is the primes.

Lazy Lists: So how do we do this in Lisp? The answer is with an idea call *Lazy Lists.* Instead of a list being made up of `cons` cells where the `car` is the first element and the `cdr` is the next `cons` cell, we delay the evaluation of the `cdr` by wrapping the next `cons` cell in a lambda. So the `car` is still the first element, but the `cdr` is a *function* which returns the next `cons` cell. Let's look at the following snippet of Lisp:

```
(setq lexical-binding t)         ; this is necessary for static scoping in elisp
(defun successor (n)             ; first we define a helper function
  (cons
    n                            ; the car of our lazy list is n
    (lambda ()                   ; the cdr of the lazy list is a function
      (successor (+ n 1))        ; which returns the next 'lazy list' cell
    )
  )
)
(defvar naturals (successor 0)) ; 0 is the first natural number
```

We now have an infinitely long list containing every natural number! Since we have to evaluate the cdr to get the next element we can't use the default list functions to access elements of this list, so let's write our own function which gets the nth element from a lazy list:

```
(defun nth-lazy-element (n ll)
  (if (= n 0)             ; if we try to access the first element
    (car ll)              ; this is just the car of the lazy list
    (nth-lazy-element     ; otherwise we want
      (- n 1)             ; the (n - 1)th element
      (funcall (cdr ll)) ; from the list that results from evaluating the cdr
    )
  )
)
```

We can test this function too:

```
(nth-lazy-element 0 naturals)  ; returns 0
(nth-lazy-element 3 naturals)  ; returns 3
(nth-lazy-element 26 naturals) ; returns 26
```

This is unsurprising if also unexciting. However there is a certain mathematical beauty in defining a "full" list of natural numbers which will never run out of successors. Play around with making lazy lists and get a feel for how they work!

Part 1 [10 pts]: Filter

The first step for defining the Sieve of Eratosthenes in Lisp, is creating a function which removes elements from a lazy list if they don't have a certain property. In particular we want to define a function `filter-ll` which takes in two arguments, a lazy list and a boolean-valued function and returns a new lazy list which only contains elements of the original lazy list which satisfy the function. For example:

```
(defvar evens (filter-ll naturals #'cl-evenp)) ; filter the even numbers from the naturals
(nth-lazy-element 0 evens)                     ; returns 0
(nth-lazy-element 10 evens)                    ; returns 20

(defvar 3mod7               ; define a all numbers which are equivalent to 3 modulo 7
  (filter-ll naturals       ; filter over all natural numbers
    (lambda (n)             ; using a function which takes a number
      (= 3 (cl-rem n 7))    ; and returns if it has a remainder of 3 when divided by 7
    )
  )
)
(nth-lazy-element 0 3mod7) ; returns 3
(nth-lazy-element 2 3mod7) ; return 17

(defun g10 (n) (> n 10))             ; define a predicate for numbers greater than 10
(defvar evens-above-10               ; define all even numbers greater than 10
  (filter-ll                         ; filter for both
    (filter-ll naturals #'g10)       ; the naturals that are greater than 10
    #'cl-evenp                       ; and are also even
  )
)
(nth-lazy-element 0 evens-above-10)  ; returns 12
(nth-lazy-element 13 evens-above-10) ; returns 38
```

Part 2 [5 pts]: Primes

Next you will use this filter function to create the Sieve of Eratosthenes. Create a lazy list `primes` of all prime numbers which uses `filter-ll` (and possibly other helper functions) to emulate the Sieve. For example:

```
(nth-lazy-element 0 primes)   ; returns 2
(nth-lazy-element 1 primes)   ; returns 3
(nth-lazy-element 9 primes)   ; returns 29
(nth-lazy-element 107 primes) ; returns 593
```

Part 3 [5 pts]: Prime Checking

Write a (very bad) primality checking function `is-prime` which takes a number and returns true if the number is prime and false (`nil`) otherwise. Your function should work by checking if the given number is in `primes` or not.

```
(is-prime 1)   ; returns nil
(is-prime 2)   ; returns t
(is-prime 7)   ; returns t
(is-prime 729) ; returns nil
```