CS 463 - Homework 5, Comparison Code (Lisp)

deadline: (+5) extra credit if turned in by Friday 3/25, 5pm. Full credit deadline is Monday, March 28th, 5pm.

Table of Contents

1 Notes

- the tester file is available here: tester5h.lisp. As before, this is how your code will be graded, assuming you don't lose points for breaking the limitations rules.
- **Lisp Version**: we are using Common Lisp, the standard language definition for which there are many implementations. I would recommend either `clisp`, because it is already available on zeus, or `SBCL`, because it generates fast code (and now is also on zeus!). As long as you don't try to get sneaky with version-dependent features or directives, there shouldn't be any issues between implementations.
  - the only difference I can think of is that the default sorting implementation is destructive and non-stable; the implementations inside of clisp/sbcl can affect testing results since the tests do not want you to be destructive. Our tests are simple enough cases that it likely will not matter, but thinking outside of this course, you should always take a moment to peek at what built-in sorting algorithm it is that you keep calling upon.
- Name your file with the pattern `netID_h5.lisp` (e.g., `gmason76_h5.lisp`).
- include a header comment with at least these details at the top of your file:

```
;;   Name: _____________
;;   G#:   _____________
;;   (other header-comments you'd like to add)
```

- a *major* theme should be writing helper functions that solve parts of the task. We can also think of the required function as being the driver function that just calls another function that needs more arguments. Turning a loop-style algorithm into recursion can often be thought of as making all variables that are used in the loop become arguments for the recursive function.
- To turn in your work, submit your single file to BlackBoard in the appropriate submission.
- Hint: most of these functions are best solved with the aid of a helper function. Look back at your loops-filled version in Python, and review how we converted loops to recursion (capturing all the variables that persisted between iterations as arguments to the recursive function).
    - better yet, turn your Haskell implementation into a Lisp implementation. By getting your algorithm into a functional style, you now only have to worry about perhaps implementing some helper functions that were available in Haskell but not in Lisp.
- Hint: if you're not done with one definition, you can just give it a `NIL` return value for full-program compilation purposes:

```
(defun fib (n) "not done yet..." NIL)
```

- you should use the interactive mode to test out the inputs. To manually test your code, try opening `clisp` in quiet mode, loading a file, and checking that e.g. `(fib 5)` gives the expected answer. Here are a few approaches:

```
demo$ clisp -q -i netID_h5.lisp
[1]> (fib 5)
5
[2]> (quit)
demo$
```

```
demo$ clisp -q
[1]> (load "netID_h5.lisp")
T
[2]> (fib 5)
5
[3]> (quit)
demo$
```

```
demo$ sbcl --load netID_h5.lisp
This is SBCL 2.0.4,  ......<snipped>......
* (trib 5)
9
* (quit)
```

Testing all functions:

```
demo$ sbcl
* (load "h5.lisp")
T
* (load "tester5h.lisp")
T
* (run-tests)
====Running TEST-AUGDENTITY====
================================
====Running TEST-ZIP-WITH====
=============================
====Running TEST-SELECT====
============================
====Running TEST-ANY====
========================
====Running TEST-CLOCKWISE====
===============================
====Running TEST-REVERSED====
==============================
====Running TEST-MAX-TWO====
=============================
====Running TEST-TRIB====
=========================
====Running TEST-COPRIME====
=============================
====Running TEST-PRIME-FACTORS====
===================================
OK
NIL
*
```

Testing individual functions: (getting back `NIL` is good - no problems found!)

```
* (load "h5.lisp")
T
* (load "tester5h.lisp")
T
* (test-collatz)
NIL
* (test-is-prime)
NIL
*
...
```

- To turn in your work, submit the single file to BlackBoard under the appropriate submission.

1.1 Don'ts

- Do not bring in any outside code (no loading/importing); just focus on lists and basic operators always available in Lisp.
  - you may lose up to all points on functions that use such imported code, and/or a flat rate penalty.
- **do not use globals** (no `*defvar*` or `*defparameter*`).
  - If you ignore this rule, you might lose up to ten points. And realistically it may cause issues with consecutive calls (e.g., what the tester will do).
- any function that serves the same, or nearly the same, purpose, may not be used. I will try to indicate obvious things to skip, but there are too many for me to catch them all. Ask on piazza, with enough time to get a response, if you feel you've got a too-easy solution.

1.2 Bonus Points!

- if you don't use any macros other than `defun`, `if`, `cond`, `let/let*`, and `lambda`, you can earn an additional (+5) bonus! Some macros/special forms to avoid (though I'm sure you can wander off and find far more) are: `LOOP`, `do`, `dotimes`, `dolist`, `for`, `progn`, etc. If you're unsure of other exciting features, give us time to respond.

1.3 Useful things

Here are some built-in functions that you may find useful. It's also something of a short-list of what I have in my own solution, so treat it as a suggested maximal set of operations, like guardrails or bowling lane bumpers.

- approved macros:
  - `if`
  - `cond`
  - `let/let*`
  - `lambda`
- basic math: +, -, *, /, mod, floor
  - `floor`, `ceiling`, `round`
- some list operations:
  - `first`, `rest`
    - `last`, `butlast`, `nth`
    - `second`, `third`, etc.
  - `cons`, `car`, `cdr`
    - also `cadr`, `cddr`, `cdddr`, etc.
  - `length`
  - `member`, `subseq`
- `T`, `NIL`
- `and`, `or`, `not`
- `equalp`
- some predicate functions:
  - `null`
  - `zerop`
  - `consp`
- `funcall` (used when we receive a function as an argument and want to call it)
- `apply`
- (don't use the built-in `reverse`, call your own `reversed` once it's complete)
- `concatenate`, `append`
- `mapcar`
- `filter`, `count`
- a few other requested/approved things:
  - some sequence-modifying things (focus on the non-destructive versions). The destructive versions require better understanding of where data is stored and shared between expressions, which makes them harder to use correctly as a novice.
    - `remove/-if/-not` (but avoid `delete/-if/-not`)
    - `substitute/-if/-not` (but avoid `nsubstitute/-if/-not`)

1.4 Distracting things

- if you find yourself using these things, you're writing further from the functional style. It's possible they may be just the solution you need to complete your algorithm as you're envisioning it, or they may be the reason you pass no extra test cases the rest of the evening… I would suggest avoiding these, as they are not required or anticipated pieces of solutions at all for this homework.
  - `setf`
  - `return`, `return-from` (use these rarely, preferably not at all)

1.5 Turning on Tail-Call Optimization

`SBCL` seems to handle tail-call optimization directly. If you want to enable it in `clisp`, you need to manually tell it that you'd like it to have a lower level of debugging information than usual, so that it doesn't need to keep around otherwise-useless frames for tracing purposes. You also need to compile the code. (See the `proclaim` and `compile` usage below). Here's a sample session. You shouldn't need it for any of the examples on our assignment, as they do not dive so deeply in a recursive fashion. But it's fun to see tail call optimization at work!

```
> (defun even (x)
 (if (= 0 x) T
 (if (= 1 x) NIL
 (even (- x 2)))))
EVEN
> (even 10)
T
> (even 10000)
*** - Program stack overflow. RESET
> (proclaim '(optimize (debug 1)))
> (compile 'even)
EVEN ;
NIL ;
NIL
> (even 10000000)
T
```

2 Functions

2.1 (defun prime-factors (n) …)

given a positive integer, return a list of its prime factors, in increasing order. Include the correct number of occurrences, e.g. `(prime-factors 24)` is `(2 2 2 3)`.

```
* (load "h5.lisp")     ;; not needed each time, just a reminder.
T
* (prime-factors 5)
(5)
* (prime-factors 50)
(2 5 5)
* (prime-factors 66)
(2 3 11)
* (prime-factors 463)
(463)
* (prime-factors 512)
(2 2 2 2 2 2 2 2 2)
*
```

2.2 (defun coprime (a b) …)

Given two positive integers, return whether they are co-prime or not. Coprime numbers don't share any divisors other than 1.

```
* (coprime 7 11)
T
* (coprime 10 21)
T
* (coprime 50 100)
NIL
* (coprime 367 463)
T
* (coprime 330 463)
T
* (coprime 222 262)
NIL
```

2.3 (defun trib (n) …)

Given a non-negative integer `n`, calculate the nth tribonnaci number (where values are the sums of the previous three items in the sequence, or 1 when there aren't enough values ahead of it). For our purposes, the sequence begins as `1,1,1,...` (with indexes 0, 1, 2, …). You need to implement a fast version (better than exponential, aim for O(n)); if your code takes some exponential amount of time to complete, you will not get credit for this function.

```
* (trib 0)
1
* (trib 1)
1
* (trib 2)
1
* (trib 3)
3
* (trib 4)
5
* (trib 5)
9
* (trib 6)
17
* (mapcar 'trib '(5 6 7 8 9 10 11 12))
(9 17 31 57 105 193 355 653)
```

2.4 (defun max-two (xs) …)

Given a list of integers `xs`, find the two largest values and return them in a list (largest first). Largest duplicates should be preserved (included) in the output. Do not modify the original list.

- *Hint*: doing one traversal and no modification is the suggested approach.

```
(max-two (list 1 2 3 4 5 1 2))
(5 4)
* (max-two (list 3 6 1 2 6 4))
(6 6)
* (max-two (list -3 -4 -5 -2 -10))
(-2 -3)
```

2.5 (defun reversed (xs) …)

Given a list of any type of values, create the list whose values are in the opposite order. Do not modify the original list. You can't call the built-in `reverse`, of course (or any similar direct reversal functionality).

```
* (reversed (list 1 2 3 4 5))
(5 4 3 2 1)
* (reversed (list 4))
(4)
* (reversed (list T NIL NIL))
(NIL NIL T)
```

2.6 (defun clockwise (grid) …)

Given a list of lists, assume it is rectangular (all rows have same length), create and return the list of lists that contains the same values, but rotated clockwise.

```
* (clockwise (list '(1 2 3) '(4 5 6)))
((4 1) (5 2) (6 3))
* (clockwise (list '(1 2 3 4 5)))
((1) (2) (3) (4) (5))
* (clockwise (list '(1) '(2) '(3) '(4) '(5)))
((5 4 3 2 1))
```

2.7 (defun any (bs) …)

Given a list of `Bool` values, return `T` if any of them are true, and return `NIL` if every single one of them is not. Do not modify the original list. You may not just do the single call to `or` with the argument list's items all unpacked into multiple arguments to `or` - do your own list-walking.

```
* (any (list NIL NIL NIL))
NIL
* (any (list T NIL T NIL))
T
* (any NIL)
NIL
```

2.8 (defun select (p xs) …)

Given a predicate function `p` as well as a list of values `xs`, create a list of all items from the argument list that pass the predicate function. Preserve ordering in your output.

- note: there are built in functions `evenp`, `oddp`, which are convenient for manual testing and exploration.
- note: you may not use `remove-if` or its cousins.

```
* (select 'evenp (list 1 2 3 4 5))
(2 4)
* (select (lambda (x) (= 1 (mod x 2))) (list 1 2 3 4 5))
(1 3 5)
* (select (lambda (x) (and (< 5 x) (< x 10))) (list 1 2 3 4 5 6 7 8 9 10 11 12))
(6 7 8 9)
```

2.9 (defun zip-with (f as bs) …)

Given a two-argument function `f` and two lists `as` `bs` of arguments to supply, create a list of the results of applying the function to each same-indexed pair of arguments from the two lists. (Zip up the two lists with the function). The answer is as long as the shortest of the two input lists.

```
* (zip-with '+ '(1 2 3) '(10 20 30))
(11 22 33)
* (zip-with '+ '(1 2 3) '(10 20 30 40 50 60))
(11 22 33)
* (zip-with (lambda (x y) (* x y))   '(1 2 3 4) '(2 4 6 8))
(2 8 18 32)
* (zip-with (lambda (x y) (* x y))   '(1 2 3 4) NIL)
NIL
```

2.10 (defun augdentity (r c) …)

Given positive ints `r` and `c` indicating number of rows and columns, create a 2D list that represents the "augmented identity matrix" with that dimension: It's the `r x c` matrix of all zeroes, except the main diagonal is all ones.

```
* (augdentity 3 3)
((1 0 0) (0 1 0) (0 0 1))
* (augdentity 3 5)
((1 0 0 0 0) (0 1 0 0 0) (0 0 1 0 0))
* (augdentity 5 3)
((1 0 0) (0 1 0) (0 0 1) (0 0 0) (0 0 0))
* (augdentity 2 2)
((1 0) (0 1))
```