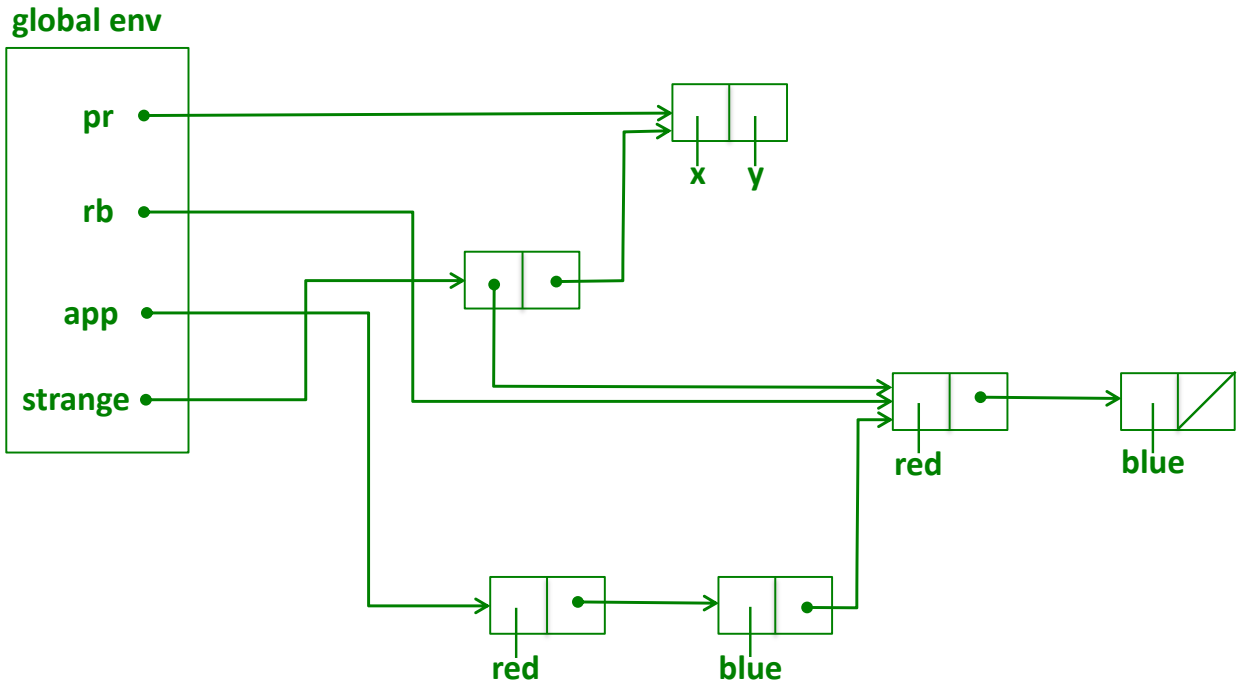
CSE 413 23sp Midterm Exam, May 1, 2023 Sample Solution

Question 1. (18 points) Lists & things. Suppose we have the following definitions in a Racket program:

```
(define pr (cons 'x 'y))
(define rb (list 'red 'blue))
(define app (append rb rb))
(define strange (cons rb pr))
```

(a) (12 points) Draw a diagram showing the combined results of evaluating these definitions together in the given order in a newly reset Racket environment.



Note: The most common error on this problem was diagraming each of the defined values independently of each other. All of the cons cells are part of the same global heap memory and they are connected as shown here.

(b) (6 points) What values are displayed if `pr`, `rb`, `app`, and `strange` are printed by Racket?

`pr:` **`'(x . y)`**

`rb:` **`'(red blue)`**

`app:` **`'(red blue red blue)`**

`strange:` **`'((red blue) x . y)`**

Question 2. (12 points, 6 points each) Local bindings. Suppose we have the following bindings at the top-level of a Racket program:

```
(define blue 2)
(define red 3)
(define green 4)
(define yellow 5)
```

If we print the expression `(red green blue yellow)` after these definitions, the output is `(3 4 2 5)`.

For each of the following two expressions, write the output that is produced if we evaluate the given expression in the global environment shown above.

(a)
```
(let ([green 6]
      [red (* green blue)])
  (list red green blue yellow))
```

`'(8 6 2 5)`

(b)
```
(let* ([blue (+ green 1)]
       [green 7]
       [yellow (* blue (+ red green))])
  (list red green blue yellow))
```

`'(3 7 5 50)`

Question 3. (16 points) (programming with lists) Write a Racket function `nitems` whose argument is a list. The function should return the number of items contained in the list. If the list contains other nested lists, the number of items in those nested lists should be added to the total.

You should not define any additional (auxiliary) functions, local or not, and your solution does not need to be tail-recursive. You may assume that all lists are proper lists with a `'()` (i.e., `null`) at the end.

Hint: you will likely want to use Racket's `null?` function in your code, and one or both of `pair?` or `list?` will be very useful. Examples:

```
(nitems '())        => 0
(nitems '(a b c))   => 3
(nitems '((a b) c d (e (f g))))  => 7
(nitems '(("recursion" 42) "is" #t ("our" (secret "friend")))) => 7
```

(Sample solution is 3 lines – you don't need to match that, but it might give some idea of what to expect.)

```
(define (nitems lst)    ;; write your code below
  (cond [(null? lst) 0]
        [(list? (car lst)) (+ (nitems (car lst)) (nitems (cdr lst)))]
        [else (+ 1 (nitems (cdr lst)))]))
```

Note: `pair?` could have been used instead of `list?` in the middle case of the conditional statement. Either one would return true for a nested proper list.

Question 4. (18 points) Tail recursion. Write a tail-recursive function `ncopies` that returns the number of times an item `x` occurs at the top level of a list. Your function should use `equal?` to determine if the list elements are equal to `x`. The function should not recursively search sublists, but, of course, if the search item `x` is itself a list, then the function should return the number of list elements that are `equal?` to `x`. Examples:

```
(ncopies 42 '())           => 0
(ncopies 42 '(17 413))     => 0
(ncopies 42 '(17 42 413)) => 1
(ncopies 42 '(17 42 #f "42" 413 42))  => 2
(ncopies 42 '(17 (42 42) "forty-two" (+ 41 1) 42))  => 1
(ncopies "forty-two" '(17 (42 42) "forty-two" (+ 41 1) 42)) => 1
(ncopies '(42) '(42 (42) 42))  => 1
```

For full credit, your implementation of `ncopies` must be tail recursive, and any auxiliary (helper) functions or other bindings must be defined inside of `ncopies`, not externally in the global environment. (sample solution is 5-6 lines)

Hint: if you decide to use an auxiliary function, it can be helpful think first about what the parameter list and specification of that function should be, then figure out how to declare it properly and call it.

```
(define (ncopies x lst)    ;; write your code below
  (letrec ([aux (lambda (nfound x lst)
                  (cond [(null? lst) nfound]
                        [(equal? x (car lst))
                           (aux (+ 1 nfound) x (cdr lst))]
                        [else (aux nfound x (cdr lst))]))])
    (aux 0 x lst)))
```

Question 5. (14 points) Higher-order functions. Write a function `npos` whose argument is a list of numbers. The function should return the number of items in the list that are > 0 (i.e., strictly positive). You may assume that the list contains only numbers and does not contain any nested sublists. Examples:

```
(npos '())              => 0
(npos `(1 413 17 42))   => 4
(npos '(-5 -1))         => 0
(npos '(-1 0 1))       => 1
(npos '(12 -15 32 -3 -4 17)) => 3
```

The catch (you knew there had to be one, right?) is that your solution cannot use `if` or `cond`. Instead, you must use higher-order functions and/or lambdas to solve the problem. In addition, you may not define any additional named helper functions or data, either in the global environment or locally. Some of the following standard functions might be useful (argument lists shown as a reminder):

```
(map procedure list)
(filter procedure list)
(foldl procedure identity list)   ;; (example: (foldl + 0 lst)
(foldr procedure identity list
(length lst)
```

Also, `(lambda (parameters) body)` might be useful.

(sample solution is 1 line, or maybe more depending on how many line breaks are included)

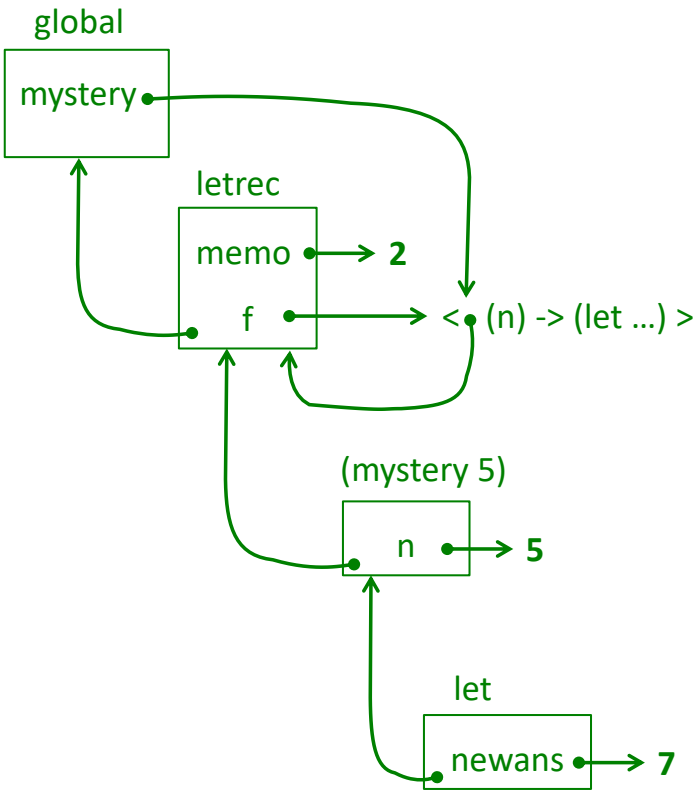
```
(define (npos lst)    ;; write your code below

  (length (filter (lambda (x) (> x 0)) lst)))
```

Question 6. (20 points) Memos. Suppose we have the following code in a racket program:

```
(define mystery
  (letrec ([memo 0]
           [f (lambda (n)
                (let ([newans (+ n memo)])
                  (begin
                    ;; >>>here!<<<
                    (set! memo newans)
                    newans)))])
    f))
(mystery 2)
(mystery 5)
(mystery 2)
```

(a) (15 points) Now suppose we start evaluating the above expressions in order. After defining `mystery`, we evaluate `(mystery 2)`, then we start evaluating `(mystery 5)`. Draw a diagram showing the bindings, environments, and closures that exist when execution reaches the point labeled `>>>here!<<<` during the evaluation of `(mystery 5)`, i.e., `(mystery 2)` has already been evaluated, and evaluation of `(mystery 5)` has reached the point right before `set!` is executed. You should only include environments and bindings that still exist when execution reaches that point. Any other environments that are not active should be omitted or crossed out.



(continued on the next page)

Question 6 (cont.) (b) (3 points) What output is produced when we evaluate all three lines of code that follow the function definition? The three lines of code are evaluated one after the other without resetting the Racket environment between evaluations. Write your answers on the blank lines after the code below:

```
(mystery 2)
```

_____**2**_____

```
(mystery 5)
```

_____**7**_____

```
(mystery 2)
```

_____**9**_____

(c) (2 points) Give a concise English description of what this function does, in the way you might explain it to a colleague or write a specification comment for this function. (Be brief – a couple of sentences should be enough.)

`mystery` produces the sum of its previous result and its current argument, and remembers that result for the next call. The "previous result" is initialized to 0 before the first call to `mystery`.

Question 7. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

The clear winner here was "streams". Opinions were divided on whether it should be included on the final exam or not. We'll see in a few weeks what happens…

(b) (1 points) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

Yes, yes, it *must* be included!!!

No opinion / don't care

None of the above. My answer is **?????**