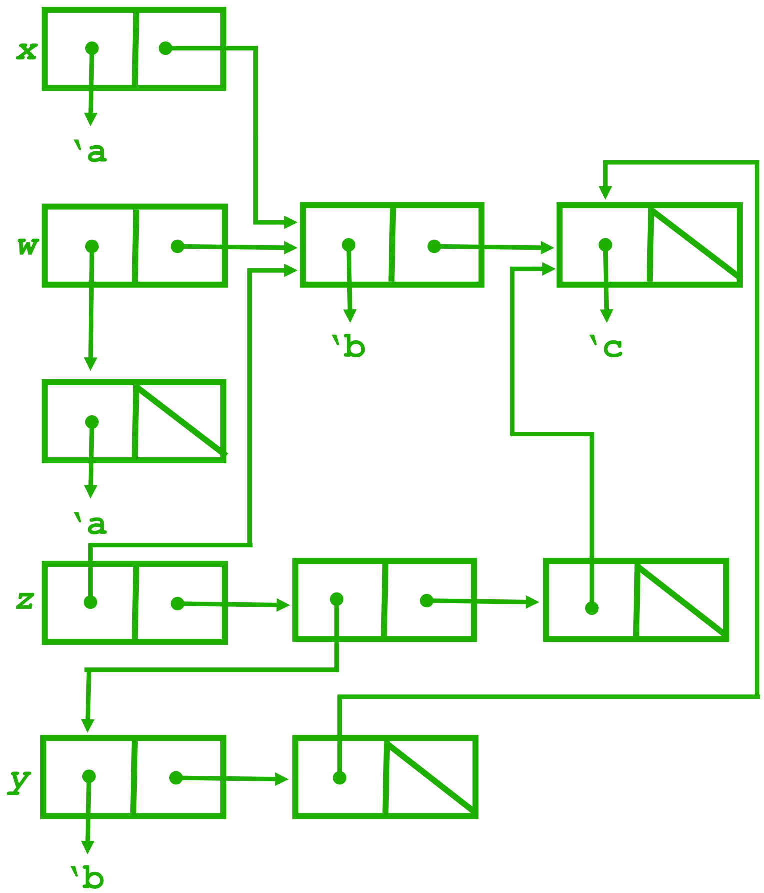
CSE 413 19wi Midterm Exam Sample Solution

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

```
(define w '((a) b c))
(define x (append (car w) (cdr w)))
(define y (list (cadr w) (cddr w)))
(define z (list (cdr w) y (cddr x)))
```

(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.



(b) (6 points) What are the values displayed if w, x, y, and z are printed by Racket?

w: `'((a) b c)`

x: `'(a b c)`

y: `'(b (c))`

z: `'((b c) (b (c)) (c))`

CSE 413 19wi Midterm Exam   Sample Solution

Question 2. (18 points) (programming with lists) (a) (9 points) Write a Racket function `concat` whose argument is a list of lists. The function should return a single list containing all the elements of the original lists. If one of the original lists contains a nested list as an element, that nested list should be included in the result as a single list element. You should not define any additional (auxiliary) functions, local or not, and your solution does not need to be tail-recursive. Hint: you will likely want to use Racket's `null?` and `append` functions in your code. Examples:

```
(concat '()) ⇒ '()
(concat '((a b) (c d))) ⇒ '(a b c d)
(concat '(() (1) (2 3) () (4))) ⇒ '(1 2 3 4)
(concat '((() w) () (x (y) ((z))))) ⇒ '(() w x (y) ((z)))
```

(Sample solution 4 lines – you don't need to match that, but it might give some idea of what's possible.)

```
(define (concat lst)     ;; write your code below
  (if (null? lst)
      lst
      (append (car lst) (concat (cdr lst)))))
```

Question 2 (cont.) (b) (9 points) Write a Racket function `flatten` whose argument is a list of nested lists. The function should return a list with the contents of the original lists, with all inner lists collapsed into a sequence of non-list elements in the original order. As before you should not define any auxiliary functions, but you *will* want to use the function `concat` from part (a). Examples:

```
(flatten '()) ⇒ '()
(flatten '((a b) (c d))) ⇒ '(a b c d)
(flatten '((() w) () (x (y) ((z))))) ⇒ '(w x y z)
```

Hint: You will find it helpful to use Racket's `list?` and `map` functions in addition to `concat`. (Sample solution 4 lines.)

```
(define (flatten lst)     ;; write your code below
  (if (list? lst)
      (concat (map flatten lst))
      (list lst)))
```

Question 3. (16 points, 8 each) For this question you are to give two implementations of a function `sum-greater` that returns the sum of all numbers in a list that are greater than some threshold. For example, `(sum-greater '(2 1 7 5 3 6) 4)` should evaluate to 18 because that is the sum of the three elements of the list (7, 5, and 6) that are greater than 4. You should assume that the list argument contains only numbers and that the threshold (the second argument) is also a number.

(a) Give an implementation of sum-greater using simple recursion and without using any higher-order functions (i.e., no `map`, `filter`, `foldl`, `foldr`, etc.). For full credit your solution must be tail recursive, and any auxiliary (helper) functions must be defined inside of `sum-greater`, not externally. (Sample solutions 7-8 lines)

```
(define (sum-greater lst thresh)    ;; write your code below
  (letrec ([aux (lambda (lst acc)
                  (cond [(null? lst) acc]
                        [(> (car lst) thresh)
                         (aux (cdr lst) (+ (car lst) acc))]
                        [else (aux (cdr lst) acc)]))])
    (aux lst 0)))
```

(b) Now give a second implementation of this function that solves the problem using higher-order Racket functions (`map`, etc.) and, if needed, using `lambda` to create anonymous functions, but does not have any conditional logic such as `if` or `cond`, and does not call any functions you have defined (even recursively). Hints: don't be intimidated by the restrictions – the solution is a fairly straightforward use of higher-order functions and should be much shorter than your version in part (a). Also, recall that `foldl` combines values in a list using a given operator and identity element: `(foldl op id lst)` and is left associative. `foldr` does the same but is right associative. (Sample solution 2 lines)

```
(define (sum-greater lst thresh)    ;; write your code below
  (foldl + 0 (filter (lambda (elem) (> elem thresh)) lst)))
```

CSE 413 19wi Midterm Exam Sample Solution

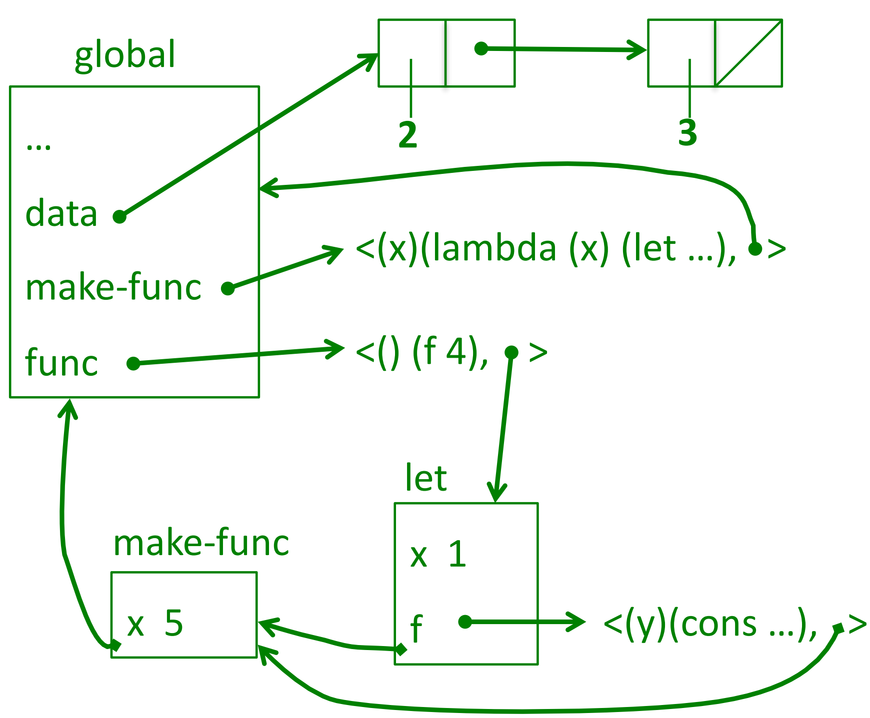
Question 4. (16 points) Environments. Suppose we execute the following definitions in Racket:

```
(define data (list 2 3))

(define make-func
  (lambda (x)
    (let ([x 1]
          [f (lambda (y)
               (cons x (cons y data)))])
      (lambda () (f 4)))))

(define func (make-func 5))
```

(a) (12 points) Draw a diagram showing the bindings, environments, and closures that exist after evaluating the above expressions. You should only include environments that remain bound after the above expressions have finished evaluating, not any other environments that existed during evaluation but are no longer active at the end.



(b) (4 points) What result is printed by Racket if we evaluate the expression `(func)` after the above definitions have been executed?

`'(5 4 2 3)`

Question 5. (16 points) Streams. Recall that we can implement a stream in Racket as a thunk (a 0-argument function) that, when called, returns a pair whose `car` is the current item in the stream and whose `cdr` is a stream (thunk) that will return the next element of the stream when used appropriately.

Write a Racket function `cycle` whose argument is a list `pattern` and that returns a stream whose elements repeatedly cycle through the element of `pattern`. When the stream reaches the end of `pattern` it should continue from the beginning and continue cycling indefinitely. Examples:

`(cycle '(a b c))` => stream representing the sequence 'a 'b 'c 'a 'b 'c …
`(cycle '(a))` => stream representing the sequence 'a 'a 'a 'a 'a …

You can assume that the pattern will contain at least one element. The elements in the pattern might be arbitrarily complex or have any type, but that should not have any effect on your solution.

Hint: you might find it useful to have a private auxiliary function in the stream closure that takes a pattern and returns an updated pattern shifted by one position, e.g., `'(a b c d)` => `'(b c d a)` takes. (Sample solutions 7-10 lines)

```
(define (cycle pattern)      ;; write your solution here

  (letrec ([shift (lambda (lst)
                    (append (cdr lst) (list (car lst))))]
           [f (lambda (lst)
                (cons (car lst) (lambda () (f (shift lst)))))])
    (lambda () (f pattern))))
```

Question 6. (16 points) Echoes of memo-like things. Suppose we have the following two function definitions, which are identical except that the order of `let` and `lambda` are reversed at the beginning of the function definition.

```
(define max-1
  (lambda (arg)
    (let ([max 0])
      (if (< arg max)
          max
          (begin (set! max arg) arg)))))

(define max-2
  (let ([max 0])
    (lambda (arg)
      (if (< arg max)
          max
          (begin (set! max arg) arg)))))
```

(a) (8 points) Now suppose we evaluate the following expressions *one after the other in the order given below*. Each expression is evaluated after any effects from the previous expressions(s) have occurred. Give the values returned by each expression as it is evaluated in sequence.

`(max-1 10)` => **10**

`(max-2 20)` => **20**

`(max-1 5)` => **5**

`(max-2 15)` => **20**

(continued on the next page)

Question 6. (cont.) (b) (8 points) Explain the results you observed in part (a). If changing the order of the `let` and `lambda` expressions did not change the results computed by `max-1` and `max-2` in part (a), explain why not. If it did make a difference, give a brief explanation of what happened. Feel free to include diagrams showing environments and closures if it helps explain things, but this is not required.

The two functions do not have the same behavior. `max-1` returns its argument or 0 if the argument is negative, while `max-2` returns the maximum value that has ever been passed to it as an argument, or 0 if it has never been called with a non-negative value.

The closure bound to `max-1` contains an ordinary `(let …)` expression and an environment pointer that refers to the global environment. Each time `max-1` is evaluated, a new `let` binding is created with `max` initialized to 0, and that environment is discarded after `max-1` finishes, even if `max` was updated by set! during evaluation of `max-1`.

When the `max-2` closure is created, however, the `lambda` expression is evaluated inside the `let` environment that initially binds `max` to 0, and the environment pointer in that closure points to that `let` environment. That means that every time `max-2` is evaluated, the same `let` environment containing the same `max` is referenced, `max` is updated if the new argument is larger than the previous value of `max`, and the most recent value of `max` is returned.