Question 1. (12 points) For each of the following, what value is printed? (Assume that each group of statements is executed independently in a newly reset Scheme environment.)

(a)
```
(define x 1)
(define y 2)
(let* ((x 5)
       (z (* x y)))
  (+ x z))
```

15

(b)
```
(define x 1)
(define y 2)
(define z (lambda (x)
            (let ((x (+ x 3)))
              (+ x y))))
(z 4)
```

9

(c)
```
(define x 1)
(define y 2)
(map (lambda (x) (+ x y)) '(3 6 -9))
```

(5 8 -7)

Question 2. (11 points) Suppose we enter the following code at the top level of a Scheme interpreter.

```
(define x 10)
(define y 11)
(define z 12)
(define f (lambda (y) (let ((x 2)) (lambda (z) (+ x y z)))))
(define mystery (f 5))
```

(a) (9 points) Give a precise (but brief) description of the value that is bound to `mystery` by this sequence of definitions. In other words, what value is returned when function `f` is applied to the value 5? If the value returned is a function closure, be sure you describe the function as well as the environment bindings (which names are bound to what) in the closure.

The result of `(f 5)` is a function closure, where the code is `(lambda (z) (+ x y z))` and the environment includes the bindings `(x 2)` and `(y 5)`. The full global environment also contains bindings for `z` and hidden bindings for `x` and `y`, but since these variables are not free in the closure they cannot be referenced in the function code, so an implementation may choose not to store these additional bindings in the closure as an optimization.

(It's also fine if you said that the code is `(+ x y z)` with parameter `y`, since we originally diagramed things that way in class.)

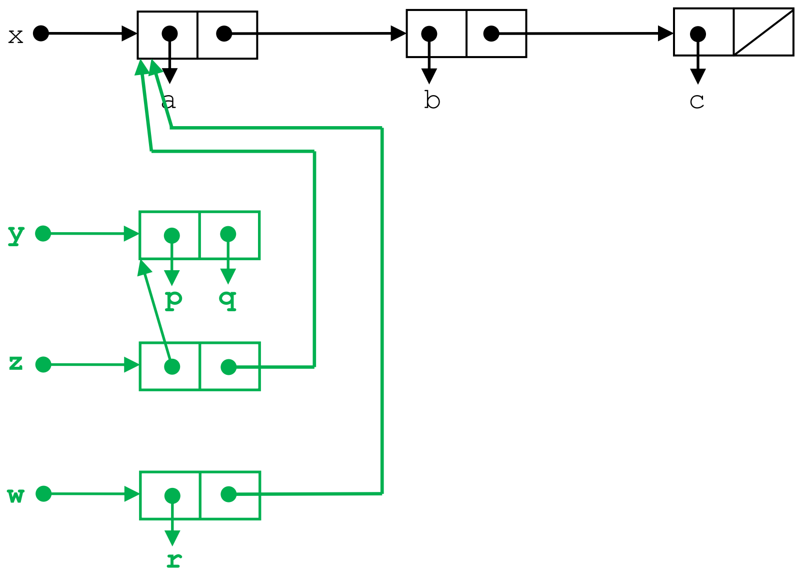
(b) (2 points) What value results if we evaluate `(mystery 6)` after executing the definitions given at the beginning of the problem?

13

Question 3. (16 points) Suppose we have the following definitions in a Scheme program

```
(define x '(a b c))
(define y (cons 'p 'q))
(define z (cons y x))
(define w (cons 'r x))
```

(a) Draw a diagram showing the result of evaluating these definitions together in the given order in a newly reset Scheme environment. The first list is drawn for you.



Note that the original list and the `(p . q)` cons cell are not copied when new cons cells are created that reference them.

(b) What are the values of `z` and `w` if these are printed by Scheme?

z `((p . q) a b c)`

w `(r a b c)`

Question 4. (18 points) The MUPL interpreter used *association lists* to represent environments. An association list is a list of `cons` pairs where the `car` of each pair is a name or key of some kind and the `cdr` is the associated value. This is a very common data structure in Scheme programming, used for many applications besides MUPL environments. For example, here is an association list that uses symbols instead of strings as the key in each pair of the list.

```
((x . 17) (y . foo) (x . 42) (bar . "secret"))
```

For this problem, write a function `(remove key alst)` that takes a value `key` and an association list `alst` as arguments. Function `remove` should return a copy of the original list except that all of the pairs in the original list whose `car` is `equal?` to `key` do not appear in the copy. For instance, if `pairs` is the example list given above, the result of `(remove 'x pairs)` should be

```
((y . foo) (bar . "secret"))
```

If `key` does not appear as the `car` of any pair in the list, the result should be a copy of the original list.

When copying all or part of the list you should not create new copies of the `cons` pairs themselves (i.e., don't duplicate `(y . foo)` and similar things). Just copy the list that references them. You should assume that the association list `alst` has the proper format and you don't need to test for that.

Your function should process the list directly and not use any library functions to search or modify the list. It does not need to be tail recursive. It should not define or use any external "helper" functions.

```
(define (remove key alst)
  (cond ((null? alst) '())
        ((equal? (caar alst) key) (remove key (cdr alst)))
        (else (cons (car alst) (remove key (cdr alst))))))
```

Question 5. (18 points) Write a tail-recursive Scheme function `(sum lst)` to compute the sum of the integer values in a list. For example, `(sum '(4 2 7))` should evaluate to 13. You may assume that the elements of the list are all integers and you do not need to handle the case if they are anything else (including nested lists of integers).
For full credit,

- Your implementation must be properly tail-recursive, and
- You may not define any additional functions or variables at top-level (i.e., `define`). You are, of course, free to create any additional functions or variables you wish nested inside the scope of `sum`, and
- You should calculate the result directly with appropriate arithmetic operations; you may not use any Scheme library functions like `fold` or anything similar.

```
(define (sum lst)
  (letrec ((aux (lambda (acc lst)
                (if (null? lst)
                    acc
                    (aux (+ acc (car lst)) (cdr lst))))))
    (aux 0 lst)))
```

It is also possible to use `(define …)` to declare the auxiliary function in the inner scope. If this was done correctly it received full credit.

Question 6. (20 points) Our head of sales has reported that we can sell 100,000 units of our MUPL interpreter to a big customer, but only if it includes one more new language feature. We need to add a "max" expression to the interpreter that computes the maximum of two integer values.

(DON'T PANIC!!! The answer is considerably shorter than the question!)

Here is the specification for the new MUPL `max` expression:

- If e_1 and e_2 are MUPL expressions, then `(make-max` e_1 e_2`)` is a MUPL expression. The value of a max expression is the larger of the two MUPL integer expressions e_1 or e_2. If e_1 and e_2 have the same value, that value is returned. If either expression is not a MUPL integer (i.e., something not created by `make-int`), then execution should be terminated with a suitable error message.

On the next page, write the code needed to add this new expression to the MUPL interpreter `eval-prog` function. Your implementation should evaluate each of the expressions e_1 and e_2 *only once* (i.e., it should not evaluate the larger expression a second time to compute the final value).

You should assume that the following structure has been added to MUPL to represent these conditional expressions:

```
(define-struct max (e1 e2))  ;; = max of expressions e1, e2
```

For reference, here are the other structures defined in the original MUPL code (most of which you probably won't need):

```
(define-struct var    (string)) ;; a variable, e.g., (make-var "foo")
(define-struct int    (num))    ;; a number, e.g., (make-int 17)
(define-struct add    (e1 e2))  ;; add two expressions
(define-struct ifgreater (e1 e2 e3 e4)) ;; if e1 > e2 then e3 else e4
(define-struct fun    (nameopt formal body))
                               ;; a recursive(?) 1-argument function
(define-struct app    (funexp actual)) ;; function application
(define-struct mlet   (var e body)) ;; (let var = e in body)
(define-struct apair  (e1 e2))  ;; make a new pair
(define-struct fst    (e))      ;; get first part of a pair
(define-struct snd    (e))      ;; get second part of a pair
(define-struct aunit  ())       ;; unit
(define-struct isaunit (e))     ;; evaluate to 1 if e is unit else 0

(define-struct closure (env fun))
```

Reminder: the Scheme function `(error "`*message*`")` can be used to terminate evaluation with the given message.

Write your code on the next page. (You can tear this page out of the exam for reference if that is convenient.)

Question 6. (cont.) Write your code to implement the new MUPL `max` expression below.

```
(define-struct max (e1 e2))  ;; new max expression

(define (eval-prog p)
  (letrec
    ((f (lambda (env p)
          (cond (...
                 ((max? p)

                   (let ([v1 (f env (max-e1 p))]
                         [v2 (f env (max-e2 p))])
                     (if (and (int? v1) (int? v2))
                         (if (> (int-num v1) (int-num v2)) v1 v2)
                         (error "MUPL max applied to non-number")))


                  )
           )
           ...
```

Question 7. (5 points) The classic mark-sweep garbage collector for Scheme examines the heap to discover which `cons` cells are currently reachable, then it reclaims all of the unreachable data by adding it to a list of available cells to be reused later. The copying garbage collector discussed in class also discovers which memory cells are in use, but it copies all of the active memory cells to new locations in memory, updating pointers to the moved cells so they correctly point to the new locations of the moved data.

It seems like the copying collector does an awful lot of extra work compared to the simple mark-sweep version. Is there any advantage to doing this? If so, what is it? (Be brief)

The main advantage is that the copying collector moves all active data so it is contiguous in memory. This reduces the memory footprint of the heap. Among other things, this should make more memory available for other programs, particularly in a paged virtual memory system.