

## CSCI2322 Midterm Answers

1. Determine the value of the following Scheme expressions. (2 points each)

a. (car '((1 2) (3 4) (5 6)))

**(1 2)**

b. (cons 9 (cons 8 (cdr '(1 2 3))))

**(9 8 2 3)**

c. (cadadr '(1 (2 3 4) 5 (6 7)))

**3**

d. ((lambda (x y) (\* x y)) 5 6)

**30**

e. (+ (cadr '(2 5 4)) (caadr '((1 2) ((5 6) 3) 4)))

**Error – I didn't mean this to be an error but it becomes (+ 5 (5 6)) which doesn't work. I needed on more a to get (+ 5 5). I gave full credit for 10 and +1 for anyone who realized that the second one was the list (5 6), though no one pointed out it was an error.**

2. Define a function named read-lists that when called will read lists input by the user and return them as elements of a list. The function stops reading as soon as the user inputs anything that isn't a list. So if the user input the following, (1 2 3) (4 5 6) (7 8 9) x, then the function would return ((1 2 3) (4 5 6) (7 8 9)).

**There are two ways to do this which produce lists in different orders.**

```
(define (read-list)
  (let
    ((elem (read)))
    (cond
      ((pair? elem) (cons elem (read-list)))
      (else '()))))
```

or

```
(define (read-list)
  (letrec
    ((helper (lambda (lst)
               (let
                 ((elem (read)))
                 (cond
                   ((pair? Elem) (helper (cons elem lst)))
                   (else lst))))
              (helper '()))))
```

3. Scheme uses lists to store just about everything, at least in the aspects of the language that we have talked about. Describe why this is the preferred style for a functional language.

**The main reason for dealing with lists is that they are easier to make immutable. Making an array or vector immutable means recreating the whole thing every time any one piece changes. Rebuilding a list only makes new cons cells and can do very little copying of data. If you don't have side effects, lists are more efficient. The fact that lists lend themselves to recursion which is the primary form of iteration in functional languages could also contribute.**

4. Euclid's algorithm is a standard procedure to quickly find the greatest common divisor (GCD) of two numbers. It is defined recursively in mathematical terms below. Define a curried function in Scheme to do this called euclid. So your function should take one argument and return a function that takes the second argument that returns the GCD.

$$GCD(i, j) = \begin{cases} i & \text{if } j=0 \\ GCD(j, i \bmod j) & \text{otherwise} \end{cases}$$

```
(define (euclid i)
  (lambda (j) (if (= j 0) i ((euclid j) (modulo i j)))))
```

**Note that his function doesn't need a letrec because the recursion goes all the way back to the outer function.**

5. Trace the following code and tell me the value returned by the last line. Show work for partial credit.

```
(define (func1 num lst)
  (cond
    ((null? lst) (cons num lst))
    ((null? (cdr lst)) (cons num lst))
    ((> (car lst) num)
     (cons (car lst) (cons (cadr lst)
                           (func1 num (cddr lst)))))
    (else (cons num lst))))

(define (func2 lst)
  (cond
    ((null? lst) '())
    ((null? (cdr lst)) lst)
    (else (func1 (car lst) (func2 (cdr lst)))))

(func2 '(2 7 1 9 3 5))
```

**This is basically an insertion sort that I broke. I broke the insert function, func1, by making it so that it skips every other element when it checks where to put the number. It actually does an OK job at first. To see this, I'm going to give the lists created by the sorting of the sublists as we come back up the call stack.**

```
(5)
(3 5)
(9 3 5)
(9 3 1 5)
(9 3 7 1 5)
(9 3 7 1 2 5)
```

**Since it is so easy to mess up what happens in Scheme with the > I'm also giving full credit if you treated the sort as ordering the other way which produces this behavior.**

```
(5)
(3 5)
(3 5 9)
(1 3 5 9)
(1 3 5 9 7)
(1 3 2 5 9 7)
```

6. The natural method of building lists and solving many other problems in Scheme is to build the answer as the functions return back up the call stack. However, this can be inefficient in some situations. What is the alternative? Site and example and explain how you solve it more efficiently.

**There are situations where a “iterative” solution works better. This type of solution builds the answer as it goes down the call stack instead of as it comes back up. We saw this in class with both the reverse function for lists and the Fibonacci numbers. In the case of reverse the problem is that cons only adds to the front of a list. So the “normal” approach is to use an append function inside of the reverse function. However, this produces code that is  $O(n^2)$ . If instead we create a helper function that takes a list as an argument, we can cons onto that list and build the reverse as we go down the call stack so it is only  $O(n)$ .**

7. You should now have seen the construction of several new “datatypes” in Scheme including the one for a student that you wrote into your assignment. Describe the process of building these in Scheme (what is put into the code that actually defines them). What are some shortcomings of doing things this way? **In Scheme we build a data type simply by agreeing on some format for a list where different elements of the list have different meanings. In our code we put functions that construct these lists and that access different parts of the list. The greatest shortcoming of this approach is one of the major shortcomings of Scheme in general, there isn't true type safety. There is no true rigidity in the structure and it is too easy to mess up. In addition, the internal organization can't really be hidden from outside code which makes this style very unsafe for larger scale programming where new datatypes are essential.**

8. Trace the following code and tell me the value returned by the last line. Show work for partial credit.

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (helper1 a)
  (* (car a) (cadr a)))

(define (helper2 a)
  (< a 10))

(define (my-func arg1 arg2 arg3)
  (cond
    ((null? arg3) '())
    ((arg1 (car arg3))
     (cons (arg2 (car arg3))
           (my-func arg1 arg2 (cdr arg3))))
    (else (cons (car arg3) (my-func arg1 arg2 (cdr arg3)))))

(my-func (compose helper2 helper1) helper1 '((1 4) (2 6) (3 2) (7 3)))
```

**This code makes extensive use of higher order functions to test your ability to use them. The first three are easy, compose, a multiply of the first two members of a list, and a test for <10. The fourth function is a bit longer, but under inspection you should see that it takes three arguments. The first is a predicate, the second is a normal function, and the third is a list. It applies the predicate to each element of the list and if the predicate is true, it applies the second argument to that element and conses it onto the result of the recursion. If it is false, it simply conses the element onto the result of the recursion. The predicate in this case multiplies two elements of a list and checks if the result is less than 10. The second function returns the product of the two elements. So for this function we get the return value (4 (2 6) 6 (7 3)). This is because  $1*4<10$  and  $2*3<10$  so we cons the product onto the list. For the others they maintain their current form.**

9. Write a function called `deep-list-replace-c` that has the ability to do a deep replace on multiple items. The function should be curried and the first argument it should take is a list of pairs, that is a list of lists where each of the internal lists has two items. The first element in each pair, if found, should be replaced by the second element. The second argument to the function is the list you will do the deep replace on. Helper functions can be written internally with `letrec` or defined as standalone functions. As an example, the call `((deep-list-replace-c '((2 4) (4 8))) '(1 2 3 4))` returns the value `(1 4 3 8)` where the 2 has been replaced by the 4 and the 4 has been replaced by the 8.

**The easiest way to do this is with two functions. The helper could be put inside as a `letrec`, but I'll write them separately to make things more apparent. The first function takes an item and the first list and returns what the item should be “replaced” with. It runs through the list checking the item against the car of each element of the list. If it finds a match, it returns the second element of that list. Otherwise, it simply returns the item. The second function simply builds a new list and uses the result of the first function for each element. Of course, the second function must be curried as requested.**

```
(define (replacement item map-list)
  (cond
    ((null? map-list) item)
    ((equal? item (caar map-list)) (cadar map-list))
    (else (replacement item (cdr map-list)))))

(define (deep-list-replace-c map-list)
  (letrec
    ((helper (lambda (list)
               (cond
                 ((null? list) '())
                 ((pair? (car list))
                  (cons ((deep-list-replace map-list) (car list))
                        (helper (cdr list))))
                 (else (cons (replacement (car list) map-list) (helper (cdr list)))))))
      helper))
```

**This only allows replacement by atoms, but is sufficient for the prompt given. One could use `let` and slightly more complex code to get around that. At least one student also took the alternate approach of repeatedly doing deep replace using each element of the `map-list` as arguments. That works equally well.**

10. Trace the following code and tell me the output of the last line. Show me work for partial credit.

```
(define (alter-list arg1)
  (letrec
    ((helper (lambda (arg2)
               (cond
                 ((null? arg2) '())
                 ((arg1 (car arg2))
                  (cons (car arg2) (helper (cdr arg2))))
                 (else (helper (cdr arg2))))))
      helper))

((alter-list (lambda (x) (> x 5))) '(1 9 7 4 6 2 8))
```

**This was intended to be the easiest of the tracing functions. It is curried and takes a predicate as a first argument, then a list as the second. For each element of the list, it keeps the element if the predicate is true on that element and effectively removes it if the predicate is false by not consing it in. So this input keeps all elements that are greater than 5. This gives us the following list.**

**(9 7 6 8)**

Take Home Extra Credit: At the end of the semester we will look briefly at the Haskell programming language. This is a derivative of ML that has some significant differences. One of the major features of Haskell that makes it stand out is “lazy evaluation”. This is the ability to not evaluate things until the value is actually needed and it allows Haskell to have things like “infinite lists”. For this problem I want you to emulate a lazy list in Scheme. You will do this by writing the functions `lazy-car` and `lazy-cdr` as well as a function `build-lazy-list`. `Lazy-car` and `lazy-cdr` do what you would expect on a value produced by `build-lazy-list`. The `build-lazy-list` function will take three arguments, a generator function, a min index and a max index. If the value passed in is `func`, it should return a lazy-list that “has” the values `((func min) (func (+ min 1)) (func (+ min 2)) ... (func max))`. The catch being that it doesn't really store all those values. It only calculates them as asked to by `lazy-car`. You can decide how to represent a lazy-list yourself, just make sure all the functions work on your representation properly. You need to e-mail me your solution by the beginning of class on Monday. (Remember that the `cdr` after the last element should return `()` so a recursive function could check for a stop condition.)