

CS51 2009 Final: Sample Solution Set

CS51 Staff
May 19th, 2009

Problem 1

(a) Impossible. When `greg` is defined initially, it must be defined as a function either of no arguments or of one argument; in either case, when the interpreter attempts to evaluate the expression, it will throw a number-of-arguments-expected error.

Note that attempting to `set!` `greg` doesn't avoid this error (try it out in DrScheme if you're not convinced). The example from section where that trick did apply was: `(if (= (f) 42) 0 (f))`. Note that the second evaluation of `(f)` is in the else branch of an if, so since if is a special form, it doesn't get evaluated until the first call is able to set! `f` to something else. Indeed, if the question had read: `(if (= (greg) 42) 0 (greg 1))`, then the solution `(define greg (lambda () (set! greg (lambda (x) 42))))` would work. However, since in Scheme the function that is going to be applied is evaluated before its arguments, the interpreter will evaluate the outer `greg` and thus require that it be a function of one argument before attempting to evaluate the inner `greg` as a function of 0 arguments.

(b) Impossible. The function `greg` needs to be defined in the global scope, before the body of the `let` is evaluated; thus, the definition of `greg` cannot reference `n`, which exists only in the local scope of the `let`.

(c) Note that `(car (car (cdr x)))` is 42 even before `m` is called. Moreover, as in b, since `greg` needs to be defined in the global scope, it can't explicitly reference `x`; and, if you tried to write a function that mutated the lambda parameter it was passed (e.g.,

```
(define greg (lambda (y) (set! y (list 3 (list 42))))), you would only be changing the local variable in the scope of the lambda. The correct solution was therefore something along the lines of:  
(define greg (lambda (x) (void))).
```

(d) `(define greg (lambda (x) (set-mcar! x 42)))`. Notice that the inner definition of `x` shadows the outer one.

(e) Impossible. The local binding of `greg` to `(lambda (x) (+ 1 x))` will override whatever the global-scope definition of `greg` that you write would otherwise be, so `(greg 0)` will return 1.

(f) `(define greg (lambda (f) f))` - add-n is exactly the function you want!

Problem 2

This question asked you to first generate an infinite stream of values representing successive terms in the infinite series expansion of $\cos(x)$.

(a)

```
(define cos-terms
  (lambda (val)
    (letrec ([loop (lambda (x i)
                    (lcons (* (expt -1 (/ i 2))
                              (/ (expt x i)
                                  (fact i)))
                          (loop x (+ i 2))))])
      (loop val 0))))
```

For part (b), note that since the series is alternating and the absolute value of each successive term is strictly less than the absolute value of the previous one, it is sufficient to simply stop adding up terms of the infinite stream as soon as the next term is less than epsilon.

(b)

```
(define cos-approx
  (lambda (x epsilon)
    (letrec ([loop (lambda (total eps stream)
                      ; Need the absolute value of the next term
                      ; here, because terms can be negative
                      (if (< (abs (lcar stream)) eps)
                          (+ total (lcar stream))
                          (loop (+ total (lcar stream)) eps (lcdr stream))))])
      (loop 0 epsilon (cos-terms x))))
```

Problem 3

(a) The equation does not hold. A counterexample:

```
(define p? (lambda (x) (zero? x)))
(define f (lambda (x) (- x 1)))
(define x '(1))
```

Then $(\text{filter } p? (\text{map } f \ x)) \Rightarrow '(0)$, whereas $(\text{map } f (\text{filter } p? \ x)) \Rightarrow '()$.

(b) In situations in which this equation holds, a compiler could take advantage of it by running the filter before the map; that way, the computational overhead of applying the map function to the elements of the list that will just be filtered out anyway can be avoided.

(c) See (a).

Problem 4

(a) Subtyping is a form of polymorphism in which there is a hierarchy of types, such that “broader” types can be used in place of “narrow” ones. In the context of OOP, subtyping is realized through subclassing. Subclasses can be used in place of their parent classes because they provide additional (or different) functionality, but still provide the interface of the superclass. They may provide a larger interface, but cannot provide a smaller interface.

The most common mistake was to confuse subtyping with subclassing. They are related, but one can implement subtyping without necessarily using classes, so it’s important to keep them separate.

(b) Classes are a collection of functions and member variable definitions that provide a blueprint for creating objects. One instantiates a class, potentially providing initialization arguments, to create an object that is an instance of that class.

(c) Inheritance is the idea that method calls made on a subclass can invoke methods of the superclass (these methods have been “inherited” by the subclass). The subclass can also override these methods to provide different functionality. Inheritance enables factoring out of common code.

The most common omission here was to forget to discuss overriding methods and the benefits that accrue from code reuse.

Problem 5

The key to this problem was to give an actual concrete example of why Joe Scheme’s function leads to impossible things.

There were two main routes one could take. The first, and most common, was to give a reduction from super-eq to the halting problem. What this means is that given super-eq, we can write a function halts? which, when given a function f and an arbitrary input i, can return whether the evaluation of f on i will terminate. This can be accomplished as follows:

```
(define (infinite n) (infinite n))

(define (halts? f i) (not (super-eq (lambda (j) (begin (f j) 0))
                                     infinite)))
```

Alternatively, it is possible to construct a contradiction directly from the definition of super-eq:

```
(define (g n) #f)
(define (f n) (super-eq f g))
```

If (super-eq f g) is true, then, (f n) evaluates to #t for all n, but (g n) evaluates to false; so, (super-eq f g) must actually be false. But then (f n) and (g n) are both #f for all inputs, so (super-eq f g) should return true.
⇒⇐

One common mistake was thinking that the halting problem is to determine whether a function halts on every input rather than just on a specific input. People also commonly would simply state that "you can use super-eq and some looping function to solve the halting problem," which is not enough of a detailed description to avoid having to actually write the halts? function. Another common pitfall was to make some kind of implicit assumptions about the way that Joe's function works (e.g., "it should be easy to pull out from the function the code that checks for the second condition and use this to solve the halting problem"). You cannot assume anything at all about the internals of this magical black-box function; after all, the whole point is that you can't actually write such a function, so how can we have any idea what kinds of tricks would be used in writing it? :)

Problem 6

There were two common solutions to this problem. The main idea is to write a function that tail-recursively puts the first list backwards onto the front of the second one; then, one can first reverse the first list, and then call the reverse-append function on the reversed first list and the original second list.

However, a common error was to assume that built-in reverse is implemented tail-recursively. This is not necessarily the case. Thus, you had to write your own tail-recursive reverse. This could be done as follows:

```
(define (treverse lst)
  (letrec ([loop (lambda (rest cur)
                  (if (empty? rest)
                      cur
                      (loop (cdr rest) (cons (car rest) cur))))])
    (loop lst '())))
```

Note that this is essentially a tailored foldl for the purposes of reversing; the following is also a tail-recursive reverse:

```
(define (treverse lst) (foldl cons '() lst))
```

Assuming that foldl is tail-recursive was fine; there's really only one way to write it, whereas there is a very natural non-tail recursive way to write reverse.

Just for completeness, a non-recursive reverse implementation would look something like:

```
(define (reverse lst)
  (if (empty? lst) lst
      (append (reverse (cdr lst)) (list (car lst)))))
```

Notice that this reverse uses append, meaning that if built-in reverse is written this way, it would be particularly bad to use it in your attempt to define append!

We can write reverse-append in one of two ways as well:

```
(define (reverse-append lst1 lst2)
  (if (empty? lst1) lst2
      (reverse-append (cdr lst1) (cons (car lst1) lst2))))
```

Or, (define (reverse-append lst1 lst2) (foldl cons lst2 lst1)).

Reverse can also be written tail-recursively as a special case of reverse-append:

```
(define (treverse lst) (reverse-append lst '()))
```

Finally, append is simply:

```
(define (append lst1 lst2) (reverse-append (treverse lst1) lst2))
```

There were a variety of common errors. The most common was miscounting the number of reverses necessary, so that the final result of the append would put one reversed list together with the other instead of the two forward lists together. Another common mistake was not realizing that the built-in reverse might not be tail-recursive. Finally, some people were confused about what tail-recursive means, and wrote a correct append that was not tail-recursive.

Note: a tail-recursive function is one in which once the recursive call has finished evaluating, there is no work remaining to be done. Writing functions in this manner is considered good style because a tail-recursive function will use constant stack space when compiled correctly.

Problem 7

The simplest way to go about this problem was to first make a mutable list of the integers from 1 to n, and then to save a pointer to the beginning of that list, walk through the whole list, and then when you got to the end, make its cdr point back to the beginning. Note that it was fine for your function to loop forever if given input less than 1, and also fine for your function to assume that the input was an integer.

```
; Makes a mutable list of the integers [k, n].
(define make-list
  (lambda (n k)
    (if (= n k)
        (mcons n empty)
        (mcons k (make-list n (+ k 1))))))

(define make-cycle
  (lambda (n)
    (letrec ([cycle (make-list n 1)]
              ; finds the end of the list, makes it point back to the beginning
              [set-end (lambda (mlist)
                          (if (empty? (mcdr mlist))
                              (set-mcdr! mlist cycle)
                              (set-end (mcdr mlist))))])
      (begin (set-end cycle) cycle))))
```

Problem 8

- (a) Total least squares; since it measures perpendicular distance to the line, it will treat x and y symmetrically.
- (b) Least median squares; since the outliers are only 45% of the data (i.e., less than half), it will ignore them.
- (c) Least squares (or total least squares), because it has a known closed-form solution.
- (d) Segmented least squares; it will compute not only the line of best fit for a given subset of the data, but also where the data should be split into new lines and how many lines to use.

Problem 9

There is a trivial solution to this problem the way that it was stated: one can simply set all of the constraint variables to be zero. This was unintentional, and the result of attempting to adapt a more complicated problem down to a situation in which it could be mapped onto Dijkstra's.

Roughly, the solution we wanted involves transforming the system of constraints into a directed graph, in which there is one node per constraint variable, and an edge between each pair of variables x_i, x_j that appear together in a constraint, with weight equal to the R.H.S. of the constraint equation in question. Then, one can add a "zero" node with an edge of weight zero leading to each node, and run Dijkstra's starting from this zero node; the shortest path to each node found should be the value given to that variable. Note that since none of these constraint values are negative, Dijkstra's will always just find a shortest path of 0, along this zero-weighted edge that we just added.

The more general problem allows constraints to be negative and then uses an algorithm called Bellman-Ford (http://en.wikipedia.org/wiki/Bellman-Ford_algorithm). Roughly, Bellman-Ford is a modification of Dijkstra's that works on graphs with negative edge weights; it also allows you to detect the existence of negative cycles in a graph, which in the framework of this problem correspond to an unsatisfiable set of constraints.