CS3L Summer 2011 Final Exam Part 2 Key

Per department policy, the finals are kept on file at the front desk of the 3rd floor of Soda Hall. (go in the main doorway on that floor, turn right, go through the glass doors). You may ask to see your final, but you won't be able to take it with you (although you can make a copy if you really want to).

Question 1:

1. This is the quintessential Thing Not To Do With HOFs. The first argument to `keep` must be a procedure, and `(not odd?)` isn't a procedure... it's a procedure call. So this will cause an **error**. To make this work, you'd have to use a lambda to generate the procedure you want:
`(keep (lambda (num) (not (odd? num))) '(1 23 456)).`

2. `cddr` is shorthand for `(cdr (cdr …))`, and just as `butfirst` of a sentence always returns another sentence (and not a word), `cdr` always returns a list. In this case, it's the list **`(revolution)`**, not just the word `revolution`.

3. This was a test of deep understanding of lambdas. The first lambda-created procedure, `(lambda (x) (x))`, takes a single argument and then calls it, with no arguments. The second lambda-created procedure, `(lambda () 'y)`, takes no arguments and returns `y`. When the second is given as an argument to the first, the second is evaluated, returning **`y`**.

4. `(list 1 '())` creates a new two-item list (the size of the list created by the list constructor is equal to the number of arguments given to that constructor) in which the first item is `1` and the second is the empty list `()`. `(cons 2 '())` creates a pair in which the `cdr` is an empty list, so this meets the definition of a list, and it's a one-item list in which the only item is `2`. `append` takes two lists and effectively changes the empty list at the end of the first one into a pointer to the second one, so we get a new three-item list: **`(1 () 2)`**.

5. Eric introduced this trick for flattening a list in one of the labs that he wrote, and many of you probably used it in the homeworks. It takes a list of sublists, and "flattens" every sublist in the list by one level, so we get **`(one (two))`**. To really understand why it works, you might have to draw out box-and-pointer diagrams.

6. This was a bit of a "gotcha", but it does test an important point: mutators such as `vector-set!` do not necessarily return meaningful values (because they don't need to), and `vector-set!` in particular just returns `okay`. It would be grand if `vector-set!` returned the new vector it had just finished making, but it doesn't. So you can't `vector-set!` the return value of another `vector-set!`, because the latter isn't a vector, and so trying to do vectory things to it will cause an **error**. I know some of you ran into this issue when you were doing HW8...

7. This was yet another of those special form / `and` / `or` / "will it or won't it run forever?" problems. `(z)` is identical to the `(run-forever)` that you're familiar with from elsewhere in the course. The `(or #t (z))` returns `#t` without ever evaluating the `(z)`, but `and` requires that all its arguments evaluate to `#t`, so the second argument must be investigated. `not` is not a special form and has to evaluate its (one) argument, resulting in a **run-forever** situation.

8. This question tested your knowledge of local environments and `define`s with multiple expressions in their body. The variable `b` is defined only within the scope of the `let`. It evaluates correctly in there, returning a value that's never used (since Scheme only returns the return value from the *last* expression within the body of a `define`), but then when it comes time to evaluate the `b` that's outside of the body of the `let`, Scheme is at a loss... it can't find a binding for it in either the current local environment (the `let` is one nested environment deeper than that) or in the global environment. So this causes an (unbound variable) **error**.

9. This procedure is supposed to determine the parity (oddness or evenness) of a number, but all that ends up happening is that `new-even?` and `new-odd?` keep foisting the problem upon each other, in a way that will **run-forever**, never getting closer to a solution. This is an example of mutual recursion, but it's a bad one, since there's no base case and the recursive calls make no net progress.

Scoring: 1 point each, all or nothing.

<u>Question 2:</u>

I-1. **False**. `accumulate` takes a procedure of two arguments, and will not work with a procedure of one argument. The way that `accumulate` works *requires* that it operate on two arguments at a time.

I-2. **True**. This is a definition from lab, and there's nothing wrong with it!

I-3. **False**, but since I decided that the wording was ambiguous, I accepted either answer. What I was thinking was: a vector itself is guaranteed to be in one contiguous block of memory, but the other things it points to (like other vectors, other lists...) are *not* guaranteed to be in that same block! But some of you might have read the question as "a vector and all of its references to those things are in the same place in memory", which *is* true.

I-4. **False**. To find the number of occurrences of a particular value in a vector, you have to check every item in the vector, because any of them could be an instance of that value. This has to take longer as the vector gets longer!

II. It's up for debate whether `filter` is a selector, but it's certainly not the right selector for sentences; using `filter` on a sentence causes a crash. `item` is a selector for words (as well as sentences). `leaf?` is a predicate function; it's not a selector, because it's not returning some piece of data stored in the Tree, even though it is returning information about the Tree. `vector-set!` is a mutator, not a selector. So you should have circled only **`item` for words**.

III. This question was inspired by some advice from a former CS3L instructor who pointed out that a good barometer of understanding Scheme syntax is: do you know the difference between `x`, `(x)`, `'x`, and `'(x)`? My version uses `car`. Here's what Scheme will print in response to these inputs:

`car: #[closure arglist...]`, which is NOT an error, as some of you thought! An error prints a big nasty message and stops whatever Scheme was doing, but this doesn't happen when a procedure is a return value of an evaluation. (If it did, Scheme would crash an awful lot, because just about any program requires Scheme to evaluate, e.g., the name of a procedure at some point.)

`(car)`: this WILL cause an error, because `car` always takes exactly one argument (a list), and it doesn't make sense to give it zero arguments – what would it do?

`'car`: this is the same as `(quote car)`; you can verify this by seeing that they're `equal?` to each other. Anything after `(quote …)` is not evaluated, and is returned as is (as a literal value).

`'(car)`: this is the shorthand we can use to create a list; in this case, the list will have one item. It's NOT a list containing the procedure `car`; it's a list containing the word `car`. (The `quote` ensures that *everything* inside the quoted list is interpreted literally and not evaluated.)

So the answers are: **III-1. B; III-2. D; III-3. C; III-4. B.**

IV: Let's trace out the process that this `append` would generate when `ls1` is `(1 2 3)` and `ls2` is `(4 5 6)`:

```
(append (1 2 3) (4 5 6))
(cons 1 (append (2 3) (4 5 6)))
(cons 1 (cons 2 (append (3) (4 5 6))))
(cons 1 (cons 2 (cons 3 (append () (4 5 6)))))
(cons 1 (cons 2 (cons 3 (4 5 6))))
(cons 1 (cons 2 (3 4 5 6)))
(cons 1 (2 3 4 5 6))
(1 2 3 4 5 6)
```

There's one step in there for every item in the first list (as we "step down" the first list), and then another step for every item in the first list as we `cons` together the new list. But the second list just sits there like a log. It never gets traversed, and it never matters how long it is. The question told you to assume that `cons` doesn't take longer to "handle" longer lists, and that's true, because all a `cons` does is create a new pair and point the two halves to the arguments. It doesn't need to fully explore those arguments.

So doubling the length of `ls1` would result in twice as many steps in the process above, and would thus **roughly double** the running time... but doubling the length of `ls2` would **not significantly change** the running time.

V. `pyramid` is a particular BST with the given properties, not a variable representing a general BST. We know that it is fully balanced and fleshed-out, and that it has a depth of four (equivalently, five "levels" of nodes with four "levels" of connections between them). Since this is a BST, the maximum element is reachable by always taking the right-child until we get to a leaf, at which point we should take the datum. So the correct expression is:

```
(datum ((repeated right-child 4) pyramid))
```

You didn't lose points for mixing up the order of the two arguments to `repeated` or for using something like `(depth tree)` in place of 4 (because several people misunderstood the problem as being about BSTs of any size), but you did lose points if you used just a mysterious undefined variable `depth` or something that wouldn't obviously evaluate to 4.

Scoring: I – 1 pt. each, all or nothing; II – 2 pts. minus 1 for omitting `item` and 1 for every other thing circled; III – 1 pt. each, all or nothing; IV – 2 pts. each, 1 pt. total for answering quadruple + double; V – 1 pt. per blank.

Question 3:

1. This was really a question about global / local environments rather than `trace`, and it's based on something that most of you have encountered several times this summer. Initially, I thought that it wasn't possible to trace helpers defined within other procedures, but then I realized that that was incorrect – the real problem is that the global STk environment is unaware of the name of the helper, so it doesn't know what you're asking it to trace! If you call trace while STk is still in the local environment where the helper lives, it works just fine. I made a Piazza post about this earlier this summer...

2. There are FIVE bugs in this procedure, and all of them are fairly common things that I've observed many times this summer. (Once all five are fixed, the procedure really does run correctly!) Only a few people found all five, and many people stopped as soon as they found one...

I. The use of `word` as both a variable name and a procedure poses a problem; the call to `word` fails because `word` has been given a new meaning by that point. Fix: Change all the instances of `word` (EXCEPT for the procedure call) to something like `wd`.

II. `length` is not the correct procedure for a word, and will cause a crash. It must be replaced with `count`.

III. The condition for branching to the base case should be when the remaining word is `halfsize` characters long, not when it's less than `halfsize` characters long. So that < should become an =. (I accepted <=)

IV. The base case is wrong – we're building up a new word, so we want the "foundation" of that to be an empty word `""`, not an empty sentence `()`.

V. The recursive call calls the outer procedure `left-half` instead of the inner `helper`. This is bad, because it causes the `let` to make new bindings every time, defeating the purpose of using it to keep track of the original size of the word (which would have been "forgotten" otherwise). So `left-half` should be replaced with `helper`.

3. We've been doing our best to stress that the presence of a helper is not what makes a procedure generate an iterative (vs. a recursive) process, and if you've been relying on some kind of (helper means iterative) mnemonic, it bit you here. This procedure generates a recursive (expanding) process. It builds up its answer by wording together a bunch of letters onto an empty word. If the original input were the word sinister, we'd build up this expanding expression, gradually:

```
(word s (helper inister))
(word s (word i (helper nister)))
(word s (word i (word n (helper ister))))
(word s (word i (word n (word i (helper ister)))))
(word s (word i (word n (word i "”))))
```

…
`sini`

Scoring: 2 points (all or nothing) for question 1, 5 points (one per fixed bug) for question 2, 2 points (all or nothing) for question 3.

Question 4:

This question asks for a procedure that returns a procedure, so there's no way around it – we'll have to use lambda! We want the `no4`-ified version of `proc` to continue to take one argument, so our lambda must begin with something like `(lambda (arg) …)`. Also, we need to compute the value of calling `proc` on that `arg`, and either return that value, or 8 if that value would be 4. But we can't do something like this:

```
(if (equal? (proc arg) 4)
     8
     (proc arg)
```

because then we'd be calling `(proc arg)` twice, which could be bad for several reasons. If `proc` is `random`, for example, and `arg` is 10, the above could easily still return 4 – let's say the first `(random 10)` comes out to 5, so we take the "false" branch of the if and return `(random 10)`... which, since it's a separate call, could easily come out to 4, defeating the whole purpose of `no4`.

The way to get around this is to use a `let` to store the computed value of `(proc arg)`. We've seen this usage of `let` many times throughout the course.

One more subtle point: since `proc` can return values that are not numbers, using = in place of `equal?` above would have caused a crash, since = only knows how to handle numbers. `equal?`, on the other hand, can deal with inputs of any type.

So the full solution is:

```
(define (no4 proc)
  (lambda (arg)
    (let ((answer (proc arg)))
      (if (equal? (proc arg) 4)
           8
           answer))))
```

The additional question is really about the substitution model. We wanted to make sure you didn't think that two independent `(random 10)`s would be plugged into the body of square independently, like so:

```
(* ((no4 random) 10) ((no4 random) 10))
```

Scheme fully evaluates arguments in a procedure call before substituting those values into the body of a procedure, so `((no4 random) 10)` has already been evaluated and boiled down to a single number before that value ever reaches the body of square. There are normally 10 possible return

values for `(random 10)` – anything from 0 to 9, inclusive – and even if you couldn't remember whether the range was 0-9 or 1-10, it didn't matter for this problem. Because of `no4`, any outputs of 4 will be replaced by 8, so there are only nine values that `((no4 random) 10)` can produce: 0, 1, 2, 3, 5, 6, 7, 8, and 9. Squaring these values doesn't make any of them the same: 0, 1, 4, 9, 25, 36, 49, 64, 81. So there are **nine** possibilities.

Scoring: 2 pts (all or nothing) for the answer to the random question; 6 pts for the code. Generally, for this one, you started with 6 points and lost points for errors (e.g. -1 for =, -2 for making multiple calls to `proc`...) or got a flat 2 points (or sometimes 1) for code that just wasn't close enough. e.g.:

6: correct
5: correct, but minor error (usually using = instead of `equal?`)
4: would be correct but makes multiple calls to `proc`
3: not close enough to the right answer, but uses a lambda in a flawed but sensible way
2: does something too wild (like `(proc)`) or otherwise seems to miss the need to use a lambda

<u>Question 5:</u>

This was intended to be an easy, straightforward question... *if* you used HOFs instead of multiple nested layers of recursion, which reinvents the wheel (three wheels, in this case) and tends to open the door to more possible errors. A theme of CS3L was "here are some different tools; use the right one for the job", and this is *definitely* a job for HOFs:

* Use `map` to go through the list of movie titles and do something to each movie title. That something is:
* Each movie title is a sentence, so use `every` (`map` also works) to go through each title and do something to each word. *That* something is:
* Use `keep` to go through each word and retain only the letters other than R.

So the skeleton of the solution is:

```
(map (lambda (title)
  (every (lambda (wd)
    (keep (lambda (ltr) …
    wd)
  title)
films)
```

The full solution is:

```
(define (film-no-r films)
  (map (lambda (title)
                (every (lambda (wd)
                               (keep (lambda (ltr) (not (equal? ltr 'r))) wd))
                       title))
       films)))
```

Possible tricky points: using the right HOFs (you can get away with another `map` in place of the second `every`, but using `filter` instead of `keep` causes a crash), keeping the three nested lambdas

straight (did you use the “lambda box” technique?)

Scoring: 8 points for a correct answer, -1 or -2 per error, depending on the severity (forgetting an entire layer like `(map (lambda (thing) …)` was a -2, for example, as was forgetting to carry out a necessary recursive call if you wrote recursively). But this question was graded leniently overall.

<u>Question 6:</u>

This problem is roughly a combination of two of the problems from lab: finding the sum of all elements in a Tree, and finding the maximum value in a Tree. However, it's harder than it looks, and a bit harder than Ian actually intended it to be, because you can't easily combine these two things into one procedure that does the right thing in all cases. If you try to write a single `best-total` procedure that percolates down a Tree and then back up, taking the `best-total` of each subtree, you run into a problem like this:

```
      4    --->    4   ---> 12          4
     / \          / \                  / \
   -5   5        3   5               -5   5
   / \                               / \
  3   2                             3   2

  take (max (3, 2, (-5 + 3 + 2))), get 3
```

in which, in the case above, you get a final `best-total` of 12, but the answer comes from a strange hybrid of nodes (bolded above) that do not constitute a valid subtree! Any code that tried to only have `best-total` call itself, without any sort of helper, ran into this problem and was unfortunately doomed to fail in cases like the above. (If you coded in this manner, and you believe that your code was correct and did avoid this problem somehow, please let me know ASAP.)

This is a relatively rare case in CS3L/61A in which the obvious, “brute force” way to tackle the problem is actually the right strategy. You have to write a procedure that totals up all the values in a subtree, and then look at all of these totals and take the maximum. The component parts look a lot like the procedures from lab that totaled Trees and found the maximum value.

```
(define (subtree-total t)
  (if (leaf? t)
      (datum t)
      (+ (datum t)
         (accumulate + (map subtree-total (children t))))))

(define (best-total t)
  (if (leaf? t)
      (datum t)
      (max (subtree-total t)
           (accumulate max (map best-total (children t))))))
```

A common mistake (in code that adopted the correct strategy, but used Tree/forest recursion instead of HOFs) was to use 0 as a base case in a helper procedure that found the maximum of best totals. This guaranteed that your overall answer would be no less than 0, which would always be

incorrect for a Tree consisting only of negative values. (Since the example Tree prominently featured negative numbers, and since this problem would be trivially easy if all the numbers were positive (the root subtree would always have the best total), I didn't think it was unreasonable to expect you to consider this case!)

Scoring:

8: correct
7: correct with a minor error
6: correct with a substantial error (e.g., I had to mess with `append` / `accumulate` / etc. to get your code to work, or you had the 0 "default" as described above)
4-5: correct with multiple minor / substantial errors
3: used the flawed strategy above, but demonstrated understanding of Trees
2: had an idea / a start, but incomplete; or, used something like the flawed strategy, but didn't demonstrate a solid enough understanding of Trees

<u>Question 7:</u>

Ugh... telegraph *troubles* indeed. I apologize for the various typos in this question, some of which were minor (and the intent should have been very clear), but others of which were enough to torpedo Question 7-2. It would have been possible to fix 7-2 during the exam, but in hindsight, dropping that question was probably the right decision anyway, since it was very difficult and many people worked until close to the end of the allotted time even without 7-2 in the mix. I'm glad we caught the problem early on (thanks, Maxime), and I don't think anyone was seriously affected by the error (anyone who was working on it at the time the question was dropped was far enough ahead that they didn't run out of time overall, I think!) But still...

The error in 7-2 was actually the same sort of error I expected you to find in Question 3: using the procedure and variable names (`parse-morse`, `w`) from an "outer" procedure instead of the correct ones from the "inner" procedure (`helper`, `remaining`). This was exactly what I told you not to do with memoization – and yet, unfortunately, the procedure really did still work as written, which is why the error wasn't caught.

Anyway, FWIW, here's what `parse-morse` is supposed to look like (starts on the next page...)

```
(define (parse-morse w)
  (let ((memovect (make-vector (+ (count w) 1) #f)))
    (define (helper rest)
      (if (empty? rest)
       '("")
       (let ((lookup (vector-ref memovect (count rest))))
         (if lookup
             lookup
             (let ((answer
                     (se  ; or append
                       (every (lambda (x)
                                (word (from-morse (first rest)) x))
                              (helper (bf rest)))
                       (if (< (count rest) 2)
                           '()
                           (every (lambda (x)
                                    (word (from-morse
                                            (word (first rest)
                                                  (first (bf rest)))
                                            x))
                                    (helper (bf (bf rest)))))))
               (vector-set! memovect (count rest) answer)
               answer)))))
    (helper w)))
```

7-1: This question was about `assoc` and how to get a value out of the list returned by `assoc`:

```
(define (from-morse ml)
  (cadr (assoc mw morse)))
```

7-3: ...and this one just tested your knowledge of list constructors and list-oriented HOFs.

```
(define (ass-backwards al)
  (map  ; every won't work! it'll flatten the list!
    (lambda (sublist) (list (cadr sublist)
                            (car sublist)))
al))
```

7-4: This question seems to have thrown a lot of people. If you look at what the results of `parse-morse` would be for various lengths of word (we'll use only dots; whether dots or dashes are involved doesn't affect the number of parsings):

```
(parse-morse '*) => (e)
(parse-morse '** => (ee i)
(parse-morse '*** => (eee ei ie)
(parse-morse '**** => (eeee eei eie iee ii)
(parse-morse '***** => (eeeee eeei eeie eiee ieee eii iei eii)
```

the pattern should seem familiar...

length of word	1	2	3	4	5
# of parsings	1	2	3	5	8

The number of items n in the list is the c+2th Fibonacci number, where the Fibonacci numbers are defined such that the first is 0 and the second is 1.

You might have seen the analogy to the “going down the stairs” problem from lab, which also boiled down to Fibonacci. At any stage of parsing our Morse word, we can choose to either read the next single character, or the next two characters. Similarly, in the staircase problem, we could always choose to either go down one stair, or go down two stairs.

Scoring: 1 point per blank for 7-1 and 7-3, 2 points (1 for a description that was on the track but not precise enough) for 7-4, 10 free happy fun points for everyone for the dropped 7-2. (But 7-2 *would* have been awesome! It's an example of memoization involving lists of values instead of just numbers or Booleans and it's really cool and... *sigh*)