CS 61AS Fall 2013 Final Solutions

Name: Brian Harvey

Login: cs61as-bh

Name of person on left (or wall): Paul Hilfinger

Name of person on right (or wall): John Denero

Circle Finals Taken:

0 1 2 3 4

Some notes:
- You are allowed a back and front cheat sheet for each quiz you have taken this semester, up to 15 sheets. Apart from these sheets, you may not consult any other materials, electronic or paper.

- There are many questions of various difficulties on each exam. Make sure to pace yourself.

- When writing procedures, don't put in error checks. Assume that you will be given the argument of the correct type.

Read and sign this:

"I have read and understood everything on this page.

I certify that my answers to this exam are all my own work, and that I have not discussed the exam questions or answers with anyone prior to taking this exam.

If I am taking this exam early. I certify that I should not discuss the exam questions or answers with anyone until after the scheduled exam time."

_____

1. A B C, it's as easy as 1 2 3

a) (2 points) Write a function `nth-alphabet` that accepts a single letter, and returns the index of that letter in the alphabet (starting from 1)

```
(define alphabet '(a b c d e f g h i j k l m n o p q r s t u v w x y z))

> (nth-letter ‘a)
1
> (nth-letter ‘b)
2
> (nth-letter ‘z)
26

(define (nth-letter alph)
    (define (helper lst)
          (if (equal? alph (first lst))
              1
              (+ 1 (helper (bf lst)))))
    (helper alphabet))
```

2. *Caesar Cipher*

(3+1+1 points) Here is a simple cipher algorithm; a Caesar Cipher. Given a word, it will return the word with each letter shifted by a certain amount. Let us consider a Caesar Cipher where you shift every letter by 2.

```
>(2-shifter ‘apple)
crrng             ; the letter 2-after a is c. The letter 2-after p is q,
etc.
```

One important part of the Caesar Cipher is that it wraps around:

```
>(2-shifter ‘yoyo)
aqaq              ; the letter 2-after y is a (it wraps around the
alphabet).
```

We are going to generalize `2-shifter`. We want to be able to `caesar-cipher` any amount between 0 and 26.

a) (3 points) Write a function `make-caesar-cipher` that accepts as its argument the shift amount and returns a function that accepts a word that caesar shifts by that amount.

```
> (define 1-shifter (make-caesar-cipher 1))
> (1-shifter ‘apple)
bqqmf
> (1-shifter 'zz)
```

```
aa
> (define 2-shifter (make-caesar-cipher 2))
> (2-shifter 'apple)
crrng
> (2-shifter 'yoyo)
aqaq
> (define 5-shifter (make-caesar-cipher 5))
> (5-shifter ‘applez)
fuuqje
```

You might find `nth-letter, alphabet`, `item`, and `remainder` useful.

```
> (item 0 alphabet)
a
> (item 25 alphabet)
z
```

Suggestion: Think about defining `2-shifter`, and generalize it.

```
(define (make-caesar-cipher shamt)
    (lambda (text)
        (if (empty? text)
            text
            (let* ( (index (nth-letter (first text)))
                    (newindex (remainder (+ index shamt) 26))
                    (newletter (item newindex alphabet)))
                (word newletter ((make-caesar-cipher shamt) (bf text)))))))
```

b) (1 point) What's the runtime of your function above?
Theta(1). The input is shamt, and all that make-caesar-cipher does is return a function.

c) (1 point) We can encrypt words using `make-caesar-cipher` defined above. Now we want to use that to cipher a whole sentence by creating a function called `cipher-sent`. Its behavior can be seen below:

```
> (cipher-sent 2-shifter ‘(bob is apple))
(dqd ku crrng)
```

Rohin easily solved this by defining `cipher-sent` as follows:

```
> (define cipher-sent every)
```

Here is the definition of `every` for reference:

```
(define (every fn sent)
      (if (empty? sent)
          ‘()
```

```
(se (fn (first sent)) (every fn (butfirst sent)))))
```

Is `cipher-sent` recursive or iterative? Justify your answer in one sentence.
recursive. Every is recursive because the recursive call is being called with another function, fn.

_____.

In Summer

3. (1+2 points) Olaf the Snowman loves summer, so whenever he sees the word `'summer` in a sentence, he put `'xoxo` before and after the word. Your job is to implement the procedure **olaf-sent** which takes a sentence as argument (which may or may not have the word `'summer`), and returns the expected sentence.

```
> (olaf-sent '(I will be doing whatever snow does in summer))
> (I will be doing whatever snow does in xoxo summer xoxo)
> (olaf-sent '(summer is awesome))
> (xoxo summer xoxo is awesome)
```

a) (1 point) You have two ways of writing **olaf-sent** – either with recursion or iteration. What is an advantage of using iteration? Explain.

Iteration is more efficient memory-wise because if you use recursion, you have to store all values, while iteration requires only constant space.

b) (2 points) Finish **olaf-sent**. Write the function iteratively.

```
(define (olaf-sent sent)
  (define (helper orig-sent new-sent)
    (if (empty? orig-sent)
        new-sent
        (if (eq? (first orig-sent) 'summer)
            (helper (bf orig-sent) (se new-sent 'xoxo 'summer 'xoxo))
            (helper (bf orig-sent) (se new-sent (first orig-sent))))))
  (helper sent '()))
```

1) “Why can’t I take a car of a binary tree? or the cdr? I know they are actually just lists!” claims Jisoo. “Well”, Mona said, “I can be evil and create a representation where every binary tree is a procedure/lambda via message passing. Actually, why don’t you do it for me!”.

a) (3 points) Define the constructors and selectors for a **binary-tree.** The representation of the tree should be a procedure which uses message-passing. Define the constructor `make-binary-tree`, and three selectors `datum, left-branch` and `right-branch`.

Note: In the original, list-representation, `the-empty-tree` is equivalent to `'()`. You can assume that the equivalent `the-empty-tree` for this lambda representation is defined for you.

```
>(define x (make-binary-tree 42 the-empty-tree the-empty-tree))  ; x is a
tree
>x               ;a tree is represented as a procedure/lambda
#closure[...]
>(datum x)
42
>(left-branch x)
the-empty-tree
>(right-branch x)
the-empty-tree

>(define y (make-binary-tree 10 the-empty-tree the-empty-tree))
>(datum y)
10

>(define z (make-binary-tree 7 x y))
>(datum z)
7
>(eq? (left-branch z) x)    ; returns the left-branch of z, which is x
#t

>(eq? (right-branch z) y)   ; returns the right-branch of z, which is y
#t

(define (make-binary-tree datum left-branch right-branch)
      (lambda (message)
            (cond ((equal? message 'datum) datum)
                  ((equal? message 'left-branch) left-branch)
                  ((equal? message 'right-branch) right-branch)
                  (else (error "Magikarp uses splash")))))
                  ;optional
```

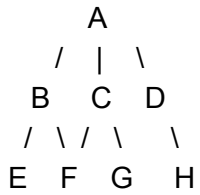
```
(define (datum tree)
      (tree ‘datum))

(define (left-branch tree)
      (tree ‘left-branch))

(define (right-branch tree)
      (tree ‘right-branch))
```

b) (1 point) “Hmmm”, ponders Jisoo. “Is that it? Do I need to do anything else?”. Mona exclaimed “Ah, silly me! I forgot to tell you to define a predicate `empty-tree?` that checks if a tree is empty or not!”. Write the `empty-tree?` predicate.

```
>(empty-tree? the-empty-tree)
#t
>(empty-tree? x)
#f
>(empty-tree? z)
#f

(define (empty-tree? tree)
      (equal? tree the-empty-tree))
```

*Disclaimer: This story is fiction. Any resemblances to places, names or events in real life are completely coincidental.

2. Nobody else in this world can solve this. Nobody else but you.
Rohin is a backpacker extraordinaire. He wants to get from one place to another but he only uses one-way roads.



The diagram above shows that Rohin can go from A to B, from A to C, from A to D. From B to E, from B to F, etc. But he can NOT go backwards (so B to A, C to A, D to A, etc is not possible).

To keep track of what roads he can take, Rohin uses get-put table.
Because he can go from A to B, he will type:
`>(put 'a 'b #t)`
Since he can't go from B to A (can't move backwards), he will type:
`>(put 'b 'a #f)`
Since he can't go DIRECTLY from A to E, he will type:
`>(put 'a 'e #f)`
Rohin put all corresponding entries in the table

a) (2 points) Write a function `next-locations` that accept as an input a location, and returns a list of locations you can go from there

```
>(next-locations 'a)
(b c d)
>(next-locations 'b)
(e f)
>(next-locations 'h)
()
```

You have access to the variable `locations` which is a list of all locations.
`(define locations '(a b c d e f g h))`

```
(define (next-locations x)
      (filter (lambda (next) (get x next)) locations))
```

b) (1 point) Write a predicate `deadend?` that takes a location as an input and returns #true if you cannot go anywhere from that location

```
(define (deadend? x)
    (null? (next-locations x)))
```

c) (3 points) Your job now is to write a function `can-go?` That accepts as inputs 2 locations and returns #t if it is possible to go from the first input to the second.

```
>(can-go? 'a 'b)
#t
>(can-go? 'b 'a)
#f
>(can-go? 'a 'e)    ;You can go from A to E by A->B->E
#t
>(can-go? 'b 'c)
#f
```

Hints and Observations:

-Don't you think the diagram looks a LOT like trees and forests?

`-next-locations` is the equivalent of `children`

`-dead-end` and `next-locations` may be helpful. You can get full credits in this question even if your `dead-end` and `next-locations` are buggy.

```
(define (can-go? start end)
      (cond ((get start end) #t)
            ((deadend? start) #f)
            (else (accumulate (lambda (x y) (or x y))
                              #f
                              (map (lambda (x) (can-go? x end) )
                                   (next-locations start))))))

;Alternative 1: mutual recursion =======
(define (can-go? start end)
      (cond ((get start end) #t)
            ((deadend? start) #f)
            (else (can-go-forest? (next-locations start) end))))

(define (can-go-forest? lst end)
      (cond ((null? lst) #f)
            ((can-go? (car lst) end)  #t)
            (else (can-go-forest (cdr lst) end))))
```

```
;Interesting solutions by students
;Alternative 2: flatmap ======
(define (can-go? start end)
      (member? end (expand-children start)))

(define (expand-children start)
      (if   (deadend? start)
            loc
            (cons loc (flatmap expand-children (next-locations start)))))


;Alternative 3 ======
(define (can-go? start end)
  (cond ((get start end) #t)
        (else (not (null? (filter (lambda (x) (can-go? x end))
                                  (next-locations start)))))))
```

1. The Final Count-up

(3 points) The class definition below is for a `counter` class.

```
(define-class (counter)
   (instance-vars (count 0) )
   (method (increment)
       (set! count (+ count 1))) )
```

Now imagine that we run that counter in parallel (like as a voting/polling system)

```
> (define x (instantiate counter))
> (parallel-execute (lambda () (ask x 'increment))
                    (lambda () (ask x 'increment))
                      ...
                    (lambda () (ask x 'increment)))
```

It is possible for the instructions to interleave and produce wrong results (imagine if each line executes at once, essentially incrementing x by one only). Pierre wants a new class, `safe-counter` that behaves similarly as a counter but can increment in parallel.

```
> (define safe-x (instantiate safe-counter))        ;note the class here
> (parallel-execute (lambda () (ask safe-x 'increment))
                    (lambda () (ask safe-x 'increment))
                      ...
                    (lambda () (ask safe-x 'increment)))
```

Define the `safe-counter` class and use either mutexes or serializers to prevent wrong results when we parallel execute like above. **Note: Avoid redundant code.**

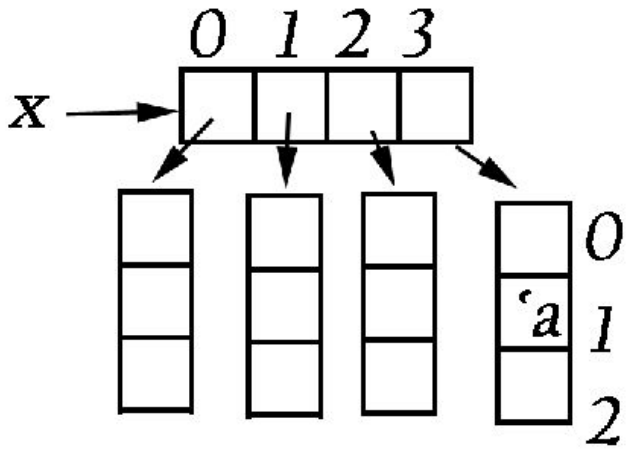
```
(define-class (safe-counter)
      (parent (counter))
      (instance-vars (mutex (make-mutex)))
      (method (increment)
            (mutex 'acquire)
            (usual 'increment)
            (mutex 'release)))

; Using serializer:
(define-class (safe-counter)
      (parent (counter))
      (instance-vars (count-protector (make-serializer)))
      (method (increment)
                  ((count-protector (lambda () (usual 'increment))))))
```

2. Empty Vectors at Empty Tables

a) (3 points) Do you remember the 2D Table? We implemented it as a list of lists. We can also implement a table as a vector of vectors. Implement `make-table` for a 2D table that is implemented as a vector of vectors. We have implemented `insert!` and `lookup` which you can use as a reference

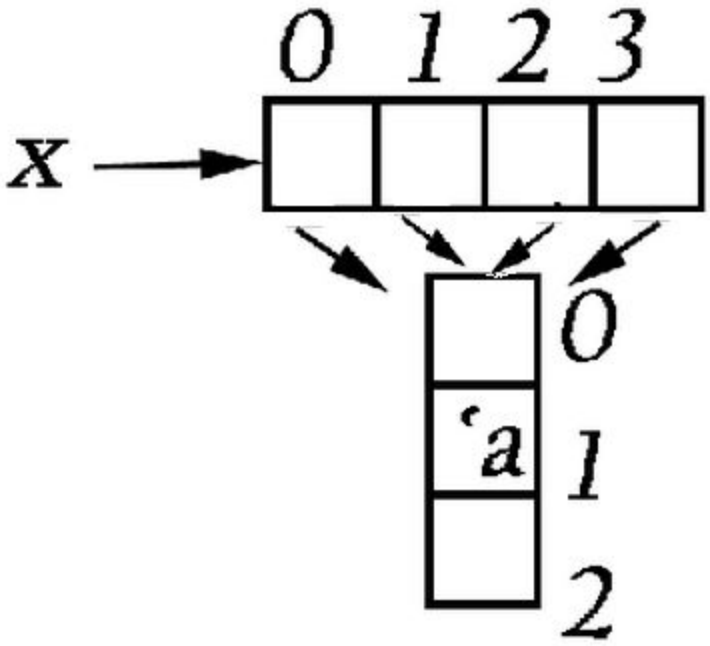
```
> (define x (make-table 4 3))      ;Creates a table of length 4 and height
3
> (insert! x 3 1 'a)               ;Put 'a at position 3,1 of table x
> (lookup x 3 1)
'a
```

Note: In the list implementation, if we try to 'lookup' something we haven't 'insert!' we return #f. You don't have to implement such behavior here.




```
(define (insert! table key1 key2 value)
   (vector-set! (vector-ref key1 table) key2 value))

(define (lookup table key1 key2)
      (vector-ref (vector-ref key1 table) key2))

(define (make-table len height)
; Constructs a vector of length 'len' that contains vectors of length
'height'

 (define output (make-vector len))
    (define (loop i)
        (if (= i len)
            output
             (begin (vector-set! output i (make-vector height))
            (loop (+ i 1)))))
```

```
    (loop 0))

;An alternative that most people used:
;Alternative 1: vector-map ======
(define (make-table len height)
      (vector-map (lambda () (make-vector height)) (make-vector len)))

;Several of you guys did this:
(define (make-table len height)
    (make-vector len (make-vector height)))

;The issue with this is that you make a SINGLE (make-vector height)
```



b) (1 point) One advantage of using a vector of vectors is that it allows constant time access for lookup. What is one **disadvantage** of using a vector of vectors for a table? (you only need to use a few words)

Disadvantages:

- fixed size
- keyed only by numbers
- some entries may be empty (our original table grows as we put more)
- Not space efficient (similar to above) in most cases
- If you want to add an element WHEN TABLE IS FULL, you have to remake a new-table with larger size. (Adding an element to table otherwise takes constant time)

3. I Have a Stream

(2 points) Let x be an infinite stream that was created from a single call to interleave. Define **EITHER** the function `uninterleave1` that gets the first stream of the two original streams, **OR** `uninterleave2` that gets the other stream. (Clearly state which one you are defining)

```
> (ss x)
(python scheme python scheme python ...)

> (ss (uninterleave1 x))
(python python python python ...)

> (ss (uninterleave2 x))
(scheme scheme scheme scheme….)


> (ss y)
(a 1 b 2 c 3 …)

> (ss (uninterleave1 y))
(a b c …)

> (ss (uninterleave2 y))
(1 2 3 …)
```

```
(define (uninterleave1 s)
    (cons-stream (stream-car s)
                        (uninterleave1 (stream-cdr (stream-cdr s)))))

(define (uninterleave2 s)
    (cons-stream (stream-car (stream-cdr s))
                         (uninterleave2 (stream-cdr (stream-cdr s)))))

;Alternative
(define (uninterleave2 s)
    (uninterleave1 (stream-cdr s)))
```

1. Sweet Mutations

(4 Points)

Brian: "Okay Hugh, type `(set! count (+ count 1))`, and for the next line, type `(set! balance (+ balance 50))`, and do the same thing for the next 5 variables!"
Hugh: "Waah! This is so tedious! I keep typing the same thing and the only thing that's different is the variable name and how much it changes by! I wish Scheme has some syntactic sugar, like `(+! count 1)` and `(+! balance 50)`!"

Winnie, who is hacking her `mceval`, overhears this and decides to add this feature in her interpreter AND a special case where if no value is specified, it increments the variable by 1 (example on the right):

```
> (define balance 0)
> (+! balance 50)
okay
> balance
50
> (+! balance (* 2 10))
okay
> balance
70
```

```
> ; An example of the special case
> (define count 0)
> (+! count)
okay
> count
1
> (+! count)
okay
> count
2
```

Make the necessary changes in MCE to implement this feature (specify what functions you add and where you are using them).

```
(define (increment? exp)
    (and    (pair? exp)
            (equal? '+! (car exp))))

(define (eval-increment exp env)
      (let* ((var (cadr exp))
             (diff (if (null? (cddr exp))
                   1
                  (caddr exp))))
          (mc-eval (list 'set! var (list '+ var diff))) env)))

;; And for those of you who know the backquote, the last line could be
  (mc-eval `(set! ,var (+ ,var ,diff)) env)

;; add this case to mc-eval, above application
(increment? exp) (eval-increment exp env)
```

2. Lazing around

(1+1+1 point) The following are conceptual questions about the Lazy evaluator discussed in lesson 13. Answer each question in four sentences or less.

a) (1 point) Consider the following code:

```
(define (crocodile x y z)
  (set! y x)
  (if (= x z)
      y
      (* x y)))

(crocodile (+ 3 5) (* 7 (/ 1 0)) (- 3 2))
```

What will Lazy evaluate this to?

64

What will MCE evaluate this to?

Error

b) (1 point) Now consider:

```
(crocodile (+ 3 5) (* 7 1) (- 3 2))
```

Is this faster in Lazy or MCE (or will they be the same)? Justify.

Faster in Lazy because (* 7 1) doesn't have to be evaluated.

c) (1 point) Why don't we use lazy evaluation all the time? In other words, what are some disadvantages to lazy evaluation?

You need to keep track of things that have been evaluated and have not, which increases overhead; Redundancy of normal order can increase the number of evaluations. If it is unmemoized, you can also be doing redundant calculations.

3. Through the Fire and Ice

(3 Points) Implement the `inject` rule, which "adds" a word in front of every word in a list. Assume the rule `same` is defined for you, such that `(same apple apple)` is satisfied but `(same apple orange)` is not.

Note: Do NOT use lisp-value

```
;;;Query Input:
(inject ice (skate rink cream) ?what)
;;;Query Output:
(inject ice (skate rink cream) (ice skate ice rink ice cream))

;;;Query Input:
(inject fire (fighter place truck)  ?what)
;;;Query Output:
(inject fire (fighter place truck) (fire fighter fire place fire truck))

(assert! (rule (inject ?word () ())))
(assert! (rule (inject ?word (?car . ?cdr)   (?out1 ?out2 . ?out3))
      (and  (same ?word ?out1)
            (same ?car ?out2)
            (inject ?word ?cdr ?out3))))

;Or, simpler solution:
(assert! (rule (inject ?word (?car . ?cdr) (?word ?car . ?rest))
               (inject ?word ?cdr ?rest)))
```