University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2008 Instructor: Dan Garcia 2008-10-29

CS3L Midterm

```
(define (recursion) (recursion))
```

Personal Information

Last name	ANSWER KEY
First Name	
Student ID Number	
Login	cs3-
The name of your TA (please circle)	Andrew Colleen DavidW DavidZ Gilbert George
Name of the person to your Left	
Name of the person to your Right	
All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS3 who have not taken it yet. ***(please sign)***	

Instructions

- Please turn off all cell phones. Remove all hats & headphones.
- You have three hours to complete this midterm. It is open book and open notes, no computers.
- Partial credit will be given for incomplete / wrong answers, so please write down as much of the solution as you can.
- **Use** `true` **instead of** `#t` **,** `false` **instead of** `#f`, since they are equivalent. Handwritten `#t` and `#f` unfortunately look too much alike…
- Feel free to write λ instead of `lambda`.
- Write the difficulty and fairness ratings in the boxes to the right and please add additional comments below.

Grading Results

Question	*Max. Pts*	*Points Earned*	*Difficulty (0=easy 5=hard)*	*Fairness (0=fair 5=unfair)*
1	**5**			
2	**7**			
3	**8**			
4	**10**			
5	**10**			
Total	**40**			

Please comment above & left:

Login: `cs3-_____`

Question 1: Grown in the USA! It was … Grown in the USA! (5 pts)

You're given the following helper functions:

- a procedure `cost` that returns the price of an item
- a predicate `food?` that returns `true` when an item is edible
- a predicate `usa?` that returns `true` when an item is made/grown in the USA

Write `cost-of-usa-food`, a function to calculate the total cost of the *food grown in the USA* from a sentence of items in a shopping list `shoplist`.

- You **may not** define any additional helper procedures
- You **may not** use `lambda` or any explicit recursion
- You **may only** use higher-order functions (as well as `cost`, `food?`, `usa?` and standard scheme built-in functions)

```
(cost 'apple)   ➔ 1     | (food? 'apple)  ➔ true  | (usa? 'apple)   ➔ true
(cost 'banana)  ➔ 3     | (food? 'banana) ➔ true  | (usa? 'banana)  ➔ false
(cost 'orange)  ➔ 5     | (food? 'orange) ➔ true  | (usa? 'orange)  ➔ true
(cost 'pencil)  ➔ 10    | (food? 'pencil) ➔ false | (usa? 'pencil)  ➔ true
```

```
(cost-of-usa-food '(apple banana orange pencil)) ➔ 6   ;; apple + orange

(define (cost-of-usa-food shoplist)

    (accumulate + (every cost (keep food? (keep usa? shoplist))))
  ____________________________________________________________ )
```

Question 2: Magical Mystery Function, step right this way… (7 pts)

```
(define (mystery x y)
   (lambda (arg)
      (first (x (word y arg)))))
```

Remember: it's <u>not</u> enough to say the domain or range is simply "a function". You have to describe the domain and range of <u>that function</u>! (…and so on if its domain / range is <u>also</u> a function)

What is the *domain* of `mystery`? (2 pts)

x is a function whose domain is a word and whose range is a word or sentence
y is a word

What is the *range* of `mystery`? (2 pts)

<u>A function whose domain is a word and whose range is a word</u>

Show a call to `mystery` **with the fewest characters in the blanks** that returns `cal`. Add *only* left paren(s) in the leftmost blank. (3 pts)

```
  ((                    (λ (w) '(cal))                    ) "")
  ((                    (λ (w) '(cal)) "")                )
_____ mystery ____________________________________ 'stanford _____
```

Left parens only

Login: `cs3-_____`

Question 3: Give me some love! XOXO (8 points)

You decide to write `love`, a function to chart how affectionate you are (i.e., what you do) with your sweetie over the course of a given `day` (day 1 is your first day together, day 2 is your second, etc.). It returns a (possibly long) word whose "alphabet" (i.e., letters used to build the word) is only: hugs (`o`), kisses (`x`), and just hanging out (`-`). The function `reverse-word` is provided for you, and does what you'd imagine it does: `(reverse-word 'abcd)` ➔ `dcba`

```
(define (love day)
  (if (< day 3)
      '-                                   ;; You just hang out for the first 2 days
      (word 'x
            (love (- day 1))
            '-
            (reverse-word (love (- day 2)))
            'o)))
```

a) What will you do on day 3? I.e., what will `(love 3)` return? If it is an error, say what the error is. If it is an infinite loop, write "it never returns". (1 pt)

x---o

b) What will you do on day 4? I.e., what will `(love 4)` return? If it is an error, say what the error is. If it is an infinite loop, write "it never returns". (2 pts)

xx---o--o

c) Now let's do some analysis of your long-term relationship. What are the *first three and last three things you do on day 9999? That is,* what are the first three and last three letters of `(love 9999)`? Fill in the blanks below. (2 pts)

x x x . . . x x o

d) `love` can return a long and seemingly random sequence of `x`s, `o`s & `-`s. For each of the following activities, circle either POSSIBLE or IMPOSSIBLE if it's ever possible to do these things someday. The first one is already done for you. (3 pts)

- (POSSIBLE) IMPOSSIBLE : `"---"` (Hang out *three* times in a row)
- POSSIBLE IMPOSSIBLE : `"----"` (Hang out *four* times in a row)
- POSSIBLE IMPOSSIBLE : `"ox"` (Hug immediately followed by a kiss)
- POSSIBLE IMPOSSIBLE : `"oo"` (Hug *twice* in a row)

Login: `cs3-_____`

Question 4 : What I need now is a cold compress on my head… (10 points)

You notice that the word `love` returns often has lots of the same characters in a row, e.g., …o--o---xx…). You decide to compress it by returning an equivalent sentence in which you've replaced all consecutive letters (including just single letters) with the consecutive number of them and the letter that's repeated. You try to write `compress`, but unfortunately, it has two bugs (you'll need to find and fix). Fill the blanks in a-e.

```
;; compress
;;
;; INPUTS   : A word, w
;; REQUIRES :
;; RETURNS  : A sentence in which every consecutive sequence of letters in w
;;          : is replaced by the number of consecutive letters and the letter
;; EXAMPLE  : Note – these are NOT NECESSARILY return values for love!
;;          : (compress 'xooooooo---x-) ➔ (1 x 6 o 3 - 1 x 1 -)
;;          : (compress '-----)         ➔ (5 -)

(define (compress w)
  (compress-helper (bf w) 1 (first w)))

(define (compress-helper w in-a-row letter)
  (cond
1   ((empty? w)
2    '())
3   ((equal? (first w) letter)
4    (compress-helper (bf w) (+ 1 in-a-row) letter))
5   (else
6    (se in-a-row
7        letter
8        (compress-helper (bf w) in-a-row letter)))))
```

a) Currently, `(compress "")` crashes. One option is to modify the code to handle it.
The *other* option is to add a REQUIRES precondition that says w can't be empty.

b) Complete the sentence below. (2 pts)

Currently, `(compress 'xxx)` returns () instead of `(3 x)`.
Replacing line # 2 with (se in-a-row letter)) fixes the bug.

c) Let's say you make the fix in part (b) above. There is one remaining bug. Fix it by replacing a single line, as you did above, and also show the *shortest sequence that triggered the bug* (and list the correct and buggy return values). After fixing both bugs in (a) and (b), `compress` **should work for all valid input**. (5 pts)

Currently, `(compress` 'xo `)` returns (1 x 1 x) instead of (1 x 1 o).

To fix it, replace line # 8 with (compress-helper (bf w) 1 (first w))))))

d) Now, **assume you've completely debugged** `compress`. If the input to `compress` has a `count` of 1000, what's the *longest* `count` of its output? 2000

e) Does `compress-helper` employ (circle one) TAIL or EMBEDDED recursion? (1 pt) EMBEDDED!

Login: `cs3-_____`

Question 5 : Number 9… Number 9… Number 9… (10 points)

A number is divisible by 9 if it is 9, or if the sum of its digits is divisible by 9.

Let's see, is the number 8888888889 divisible by 9? Well, is it 9? No, so let's check if the sum of the digits is divisible by 9. Let's see, 8+8+8+8+8+8+8+8+8+9 = 81. Ok, is 81 divisible by 9? Well, is it 9? No, so let's check if the sum of the digits is divisible by 9. Let's see, 8+1 = 9. Ok, is 9 divisible by 9? Well, is it 9? Yep! Then 8888888889 was divisible by 9! Actually, we don't really care about 9-divisibility of a general number. We first want to know *how many recursive steps a multiple of 9 took until it got to 9*. 8888888889→81→9 was 2 steps. Then want to know, for the first `n` multiples of 9 (9, 18, 27, …), **how many steps each took through that algorithm until it was 9.**

```
(define (9? n) (= n 9))
(define (9* n) (* n 9))
(add-digits 8888888889)                    ➔ 81
(repeateds-until add-digits 9? 9)          ➔ 0
(repeateds-until add-digits 9? 81)         ➔ 1
(repeateds-until add-digits 9? 8888888889) ➔ 2
(9s 1)                  ➔ (9)
(9s 15)                 ➔ (9 18 27 36 45 54 63 72 81 90 99 108 117 126 135)
(steps-until-9 15)      ➔ (0  1  1  1  1  1  1  1  1  1  2  1    1   1   1)
```

a) **Without recursion**, write `add-digits` to return the sum of the digits of its input.

```
(define (add-digits n) (accumulate + n) )
```

b) Write `repeateds-until`, which takes a function `f`, a predicate `pred?`, and an `input`, and returns the # of times `f` is called on input (ala `repeated`) until `pred?` is satisfies. E.g., if `(pred? input)` ➔ `false`, and `(pred? (f input))` ➔ `false`, but `(pred? (f (f input)))` ➔ `true`, then `(repeateds-until f pred? input)` ➔ `2`.

```
(define (repeateds-until f pred? input)
        (pred? input)
   (if  0
        (+ 1 (repeateds-until f pred? (f input)))
        ))
```

c) **Without recursion**, write `9s` that returns the first `n` multiples of 9.
Hint: first use `repeated` to generate a sentence of all the numbers from `1` to `n`, then given that answer, think about how you would generate the first `n` multiples of 9. (3 pts)

```
(define (9s n)
    (map 9*
    ((  (λ (s) (se s (+ 1 (last s)))) (- n 1)) (se 1)))
    _____ repeated _____ )
```

d) **Without recursion**, write `steps-until-9` that performs as described above. (3 pts)

```
(define (steps-until-9 n)
    (map (λ (i) (repeateds-until add-digits 9? i)) (9s n))
    _____
    _____ )
```